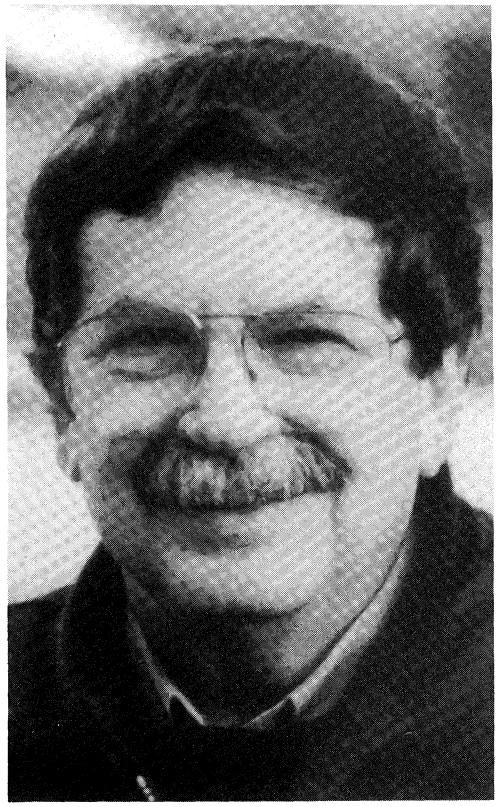
IN MEMORIAM
EUGENE L. LAWLER



Eugene L. Lawler died on September 2, 1994, at age 61, after an eight month battle with cancer. He is survived by his wife Marijke, his son Stephen, and his daughter Susan, son-in-law Matthew, and granddaughter Janna Rose Surprise. He will be dearly missed by his students, colleagues, and friends.

Gene obtained an A.M. at Harvard University in 1957 and was a Senior Electrical Engineer at Sylvania Electric Products in Needham, Massachusetts from 1959 until 1961. He went back to Harvard to obtain a Ph.D. in 1962. He taught at the University of Michigan in Ann Arbor from 1962 until 1970 and the University of California at Berkeley from 1971 until his death. He combined an illustrious career of highly influential research with a history of dedicated service to both universities. Throughout his career, Gene was an active member of the theoretical computer science community; he served on the editorial board of *SIAM Journal on Applied Mathematics* (1968–1972) as well as *SIAM Journal on Computing* (1972–1980).

For more than 30 years, Gene Lawler studied algorithmic issues in combinatorial optimization. His contributions were fundamental in giving the discipline the breadth and depth it has now attained. Of all of his work, his textbook *Combinatorial Optimization: Networks and Matroids* (1976) has had the most pronounced impact. It brought together the most important results in the area and is notable for its lucid writing style. It gave new clarity to both well-understood and not so well-understood results, brought the reader to the forefront of the field, and made the challenges of the future both apparent and accessible. It is one of the classics of the area and is as useful today as the day it was written. Another book, *The*

Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization (1985), which he edited with three younger colleagues, also became a benchmark reference.

It is hard to separate Gene's contributions as an expositor from those as a researcher. His great gift in investigating a computational approach to a problem was his ability to extract the essential difficulty, achieve a deeper insight, and solve a more general problem in a simpler way. To some extent, his expository talent came from the relative difficulty he had in absorbing new ideas. In order for him to understand other people's work, especially when it was written in a complicated way, he often had to wrestle with it to arrive at a better understanding and a simplification of the result.

Gene's papers on branch-and-bound (with D. E. Wood) and dynamic programming (with J. M. Moore) are classics; the former, in fact, was selected as a citation classic in 1987. Rather than introducing radically new techniques, both papers brought a new level of usefulness and understanding to important algorithmic paradigms. Since the mid 1970s, Gene was particularly interested in sequencing and scheduling. Prior to his work, the area was a rather unmathematical hodgepodge, with little systematic understanding of the types of methods and techniques that could be used most effectively. Gene's work greatly stimulated and unified the area. His main unfinished project is the completion of a graduate textbook on scheduling. He made significant contributions to a wide variety of questions in the area of combinatorial optimization, which were influential in both the operations research and computer science communities. Most recently, he had turned his attention to combinatorial problems in computational biology, which is an area of growing importance.

As this summary of Gene's research contributions might suggest, he was a phenomenal educator. He could provide the intuition that made difficult results easily accessible. His appreciation of the difficulties of absorbing ideas to the point that one can go beyond them in some original way helped to make him a great advisor. He was constantly available for every new idea and always ready to interest his students in whatever he was currently thinking about.

Gene had an enormous influence on the atmosphere of the Computer Science Division at Berkeley. He never lost sight of the mission of a university and never backed away from difficult tasks. Gene was the social conscience of the Division. He helped the individual student fight the bureaucracy, reformed what the university taught and to whom it taught it, and made the university a more humane and more stimulating place to study. Last year he was awarded the Berkeley Citation, the campus' highest accolade.

Gene Lawler was a remarkable man who was ready to discuss intelligently almost any current issue, and who did so in a thought-provoking way. We will all miss him very much.

SIAM J. COMPUT.
Vol. 24, No. 1, pp. 3–29, February 1995

MATRIX TRANSFORMATION IS COMPLETE FOR THE AVERAGE CASE*

ANDREAS BLASS† AND YURI GUREVICH‡

Abstract. In the theory of worst case complexity, NP completeness is used to establish that, for all practical purposes, the given NP problem is not decidable in polynomial time. In the theory of average case complexity, average case completeness is supposed to play the role of NP completeness. However, the average case reduction theory is still at an early stage, and only a few average case complete problems are known. The first algebraic problem complete for the average case under a natural probability distribution is presented. The problem is this: Given a unimodular matrix X of integers, a set S of linear transformations of such unimodular matrices and a natural number n, decide if there is a product of $\leq n$ (not necessarily different) members of S that takes X to the identity matrix.

Key words. average case, decision problems, unimodular matrices, reduction, randomization

AMS subject classifications. 68Q15, 60C05, 15A36, 15A25

1. Introduction. The theory of NP completeness is very useful. It allows one to establish that certain NP problems are NP complete and therefore, for all practical purposes, not decidable in polynomial time (PTime). One way around the NP completeness phenomenon is to consider the given NP problem together with an appropriate probability distribution and seek a decision algorithm that runs quickly on average. This works very well for some problems (see [GS], for example), but some other randomized decision problems appear too difficult even on average. It would be very useful to generalize the theory of NP completeness to be able to establish that certain randomized decision problems are, for all practical purposes, not decidable quickly on average. This is the motivation for the theory of average case completeness.

Before we plunge into this theory, let us review briefly the NP completeness theory. The idea is that PTime algorithms are considered easy. In particular, PTime decidable NP problems are considered easy. One says that an NP problem Π_1 reduces to an NP problem Π_2 if there is a PTime algorithm R from instances of Π_1 to instances of Π_2 that takes positive instances to positive instances and negative instances to negative instances. Such an R is a many-one PTime reduction from Π_1 to Π_2. A PTime decision algorithm A for Π_2 gives rise to the PTime decision algorithm $R \circ A$ for Π_1. A decision problem is hard for NP (via many-one PTime reductions) if every NP problem reduces to it (by means of a many-one PTime reduction). A decision problem is complete for NP if it belongs to NP and is hard for NP. Most known natural NP problems are either PTime decidable or NP complete.

The theory of average case completeness was pioneered by Levin in [Le]. Levin replaced NP with the class RNP of NP problems with so-called PTime computable probability distributions [Le], [Gu1]. He generalized PTime computability to computability in time polynomial on average (APtime computability) and defined many-one PTime reductions of RNP problems. Then he established that a bounded version of the known tiling problem together with a natural probability distribution is complete for RNP via many-one PTime reductions. (That is, the randomized tiling problem belongs to RNP and every problem in RNP reduces to it via many-one PTime reductions.) Another RNP problem, implicitly present in [Le], is bounded halting, a bounded version of the standard halting problem together with natural probability distribution; this problem is explicitly defined and proved complete in [Gu1].

*Received by the editors May 18, 1992; accepted for publication (in revised form) August 31, 1993.

†Mathematics Department, University of Michigan, Ann Arbor, Michigan 48109-1003 (ablass@umich.edu). The work of this author was partially supported by National Science Foundation grants DMR 88-01988 and DMS-9204276.

‡Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, Michigan 48109-2122 (gurevich@umich.edu). The work of this author was partially supported by National Science Foundation grants CCR 89-04728 and CCR 92-04742 and Office of Naval Research grant N00014-91-J-11861.

Some progress has been achieved in the meantime. In particular, the restriction to PTime computable distributions was liberalized [BCGL]; Levin's complete problems remained complete (we return to the issue of the liberalized RNP later in this introduction). The reduction theory has been revised [Gu1], [BCGL], [VL], [BG1], [BG2]. In particular, deterministic reductions, shown insufficient in [Gu1], have been replaced in [VL] by randomizing reductions; in other words, reduction algorithms have been allowed to flip coins. This tuning up of the reduction theory continues in this paper. In §§2–5 of this paper, we describe the current state of the reduction theory in full detail and in particular define a clean notion of randomized many-one reductions of randomized decision problems.

A number of additional natural RNP complete problem have been found [Gu1], [VL], [Gu2], [VR], but that number is still very small. Moreover, none of the known complete problems, however natural they are, arose in applications. All of them were designed especially for the purpose of finding additional average-case complete problems. Why have not more problems been found? It is possible that the reduction theory must be tuned up further. It is certainly true that establishing average-case completeness is much more difficult than establishing worst-case completeness; the range of an average-case reduction of a problem Π_1 to a problem Π_2 cannot comprise only very, very special instances of Π_2. What should be done? Consider the theory of undecidability or NP completeness. In either case, a rich collection of complete problems (complete for recursive enumerability via recursive reductions or NP complete, respectively) has been accumulated that are convenient for reductions to other problems. We need, it seems, to accumulate a rich collection of various average-case complete problems with the hope that these problems will be useful for further reductions. In this connection, Levin challenged Gurevich (who started his career an algebraist) to find an average-case complete problem of algebraic character. Such a problem was found in [Gu2]; this paper is a full version of the extended abstract [Gu2].

The matrix decomposition problem involves linear transformations of unimodular matrices. The *modular group* is the multiplicative group $\mathrm{SL}_2(Z)$ of two-by-two integer matrices of determinant 1 (*unimodular matrices*). The notion of linear transformation of $\mathrm{SL}_2(Z)$ does not seem to make sense because $\mathrm{SL}_2(Z)$ is not closed under addition, but this difficulty is not serious. Define a *linear transformation* of $\mathrm{SL}_2(Z)$ to be a function T from $\mathrm{SL}_2(Z)$ to $\mathrm{SL}_2(Z)$ such that $T(\sum X_i) = \sum T(X_i)$ whenever all the X_i and $\sum X_i$ are unimodular matrices. We show in §9 that a linear transformation T of $\mathrm{SL}_2(Z)$ uniquely extends to a linear transformation of all two-by-two integer (or even complex) matrices; this gives rise to the standard representation of T by a four-by-four integer matrix. Moreover we will describe a simple (certainly PTime) test to determine when a given four-by-four integer matrix represents a linear transformation of $\mathrm{SL}_2(Z)$. Identify linear transformations with the four-by-four integer matrices representing them.

Now we are ready to define our randomized decision problem. An instance of Matrix Transformation comprises three components: a unimodular matrix X, a finite set $\mathcal{S}$ of linear transformation of unimodular matrices, and a natural number n. The corresponding question is whether there exists a linear transformation $T \in \mathcal{S}^n$ such that $T(X)$ is the unit (or identity) matrix. Here $\mathcal{S}^n$ comprises products $T_1 \cdots T_m$ where $m \leq n$ and each $T_i \in \mathcal{S}$.

Define the size of an integer matrix (whether it is two-by-two or four-by-four) to be the length of the binary representation of the maximal absolute value of the entries. The size of an instance $(X, \mathcal{S}, n)$ is n plus the size of X plus the sum of the sizes of all members of $\mathcal{S}$.

The probability distribution on the instances is rather natural. The three components of a random instance $(X, \mathcal{S}, n)$ are chosen independently. The integer component n is chosen with respect to the default probability function $1/n(n+1)$. (The choice of the default distribution does not matter much [Gu1].) To choose the unimodular component X, first choose a positive

integer k with respect to the default distribution and then choose X randomly (with respect to the uniform probability distribution) among all unimodular matrices of size k. An auxiliary probability distribution on linear transformations is defined similarly; linear transformations of the same size have the same probability and the probability to have size k equals the default probability of k. Finally, the probability of $\mathcal{S}$ is proportional to the product of the probabilities of the members of $\mathcal{S}$. This completes the definition of Matrix Transformation.

We reduce Bounded Halting to Matrix Transformation and this way prove Matrix Transformation is complete for RNP via randomized many-one reductions. It remains complete under various restrictions on the cardinality of $\mathcal{S}$ and/or the number n; see §10 in this connection.

Actually, we prove only that Matrix Transformation is hard for RNP. It is obvious that (the unrandomized version of) Matrix Transformation is NP. Checking that the probability distribution is PTime is routine and we ignore it. We have already mentioned that the definition of RNP has been liberalized in [BCGL] by allowing more general distributions called *samplable*. Impagliazzo and Levin proved that every NP search problem with samplable distribution reduces via many-one PTime computable reductions to an NP search problem with PTime computable distribution [IL] (see also [BG2]). In an appropriate sense a search problem with a PTime computable distribution reduces to a decision problem with PTime computable distribution [BCGL]. Thus, Matrix Transformation is hard for the class of NP search problems with samplable distributions. Our reduction of Bounded Halting to Matrix Transformation can be easily modified to obtain a many-one randomized reduction of the search version of Bounded Halting to the search version of Matrix Transformation. Thus, the search version of matrix transformation is complete for the class of NP search problems with samplable distributions. The question remains whether Matrix Transformation (or Bounded Halting) is complete for the class of NP decision problems with samplable distributions. In the usual NP theory, decision problems are easily reducible to search problems. The situation is different in the average-case theory. We intend to consider these issues in [BG3].

A simpler version of Matrix Transformation is obtained by making $\mathcal{S}$ just a set of unimodular matrices and asking whether another unimodular matrix X can be represented as a product of at most n matrices from $\mathcal{S}$. This bounded version of the classical membership problem [Mi, p. 511] for $SL_2(Z)$ is NP complete; see § 11 in this connection. However, we do not think that the naturally randomized version of the bounded membership (§11) problem is complete for the average case. Indeed, there are indications that it is solvable in time polynomial on average. However, Venkatesan and Rajagopalan proved that the same problem for higher-dimension matrices is complete for the average case [VR].

Since we deal almost exclusively with randomized decision problems, the term "decision problem" will usually mean "randomized decision problem"; similarly, the term "search problem" will usually mean "randomized search problem."

2. Domains. As a general framework for the study of average case complexity, we use domains [Gu2], [BG1], [BG2].

DEFINITION 2.1. *A* domain X *consists of*

- *An underlying set, the* universe *of* X*, often called* X *as well, comprising strings in some alphabet* Σ_X;
- *A size function, assigning to each* $x \in X$ *a positive integer* $|x| = |x|_X$ *called the* size *of* x; *and*
- *A probability distribution* $\mathbf{P}_X$ *on* X.

We require elements of the domain to be strings in order to use the usual computation model based on the Turing machine. In the rest of the paper, an algorithm is a Turing machine. Traditional concepts of (worst-case) complexity are defined by means of the size function $|x|$. Concepts of average-case complexity are defined by averaging with respect to the probability

distribution $\mathbf{P}_X$. As was pointed out by Levin [Le] and discussed in some detail in [BCGL], [Gu1], the most obvious definition of the concept "polynomial time on average" has inappropriate consequences, and some care is needed to obtain a suitable definition. We use the following definition due to Levin [Le], as modified in [BG1] to allow ∞ as a value.

DEFINITION 2.2. *Let T be a function from a domain X to the interval $[0, \infty]$ of the real line augmented with ∞. T is* linear on average *if $T(x)/|x|$ has finite expectation,*

$$\mathop{\mathbf{E}}_{x} \frac{1}{|x|} T(x) = \sum_{x} \mathbf{P}_X(x) \frac{1}{|x|} T(x) < \infty,$$

and T is polynomial on average, *abbreviated AP, if it is bounded by a polynomial of a function that is linear on average. In other words, T is AP if, for some $\varepsilon > 0$,*

$$\mathop{\mathbf{E}}_{x} \frac{1}{|x|} (Tx)^{\varepsilon} = \sum_{x} \mathbf{P}_X(x) \frac{1}{|x|} (Tx)^{\varepsilon} < \infty\,.$$

We use the convention that $0 \cdot \infty = 0$; thus, an AP function can take the value ∞ but only at points of probability 0.

LEMMA 2.3 [Gu1].

- *If $\mathbf{E}(|T(x)| \mid |x| = l)$ is bounded by a polynomial of l, then T is AP.*
- *On any domain, the collection of AP functions is closed under addition and multiplication.*

DEFINITION 2.4. *A (deterministic) algorithm, taking elements of a domain X as inputs, is* polynomial time on average *or* AP time *if its running time on input x is an AP function of x.*

We consider the running time to be ∞ if the algorithm fails to terminate, so an AP time algorithm must terminate on all inputs of nonzero probability. In general, we take the point of view that instances of zero probability do not matter. By following that line consistently, which we try to do, we often have the luxury of throwing elements of zero probability out and supposing, when convenient, that no element of the domain X in question has zero probability or throwing elements of zero probability in and supposing, when convenient, that every string over Σ_X is an element of X. However, we do not go so far as to identify two domains if one of them is obtained from the other by eliminating some elements of zero probability.

In the case of domains with finitely many elements, it would be natural to call a domain uniform if all elements have the same probability. This definition makes no sense in the case of infinite domains, which is the only case of interest to us. Another natural way to define uniform domains requires a default probability distribution on positive integers; it is customary to assign the probability $1/n(n+1)$ to a positive integer n. The choice of the default probability distribution does not matter much; see [Gu1] in this connection.

DEFINITION 2.5. *PI is the domain of positive integers such that $|n| = n$ and the probability of any n is $1/[n(n+1)]$. The probability of a number n in PI is called the* default probability *of n.*

DEFINITION 2.6. *A domain is* uniform *if it has a finite number of elements of any given size, all elements of a given size have the same probability, and $\mathbf{P}\{x : |x| = n\}$ equals the default probability of n.*

Here are two examples of uniform domains.

DEFINITION 2.7. *BS is the uniform domain of nonempty binary strings where the size of a string is its length. FRACTION is the uniform domain of fractions a/b where a, b are relatively prime positive integers, $a \le b$ and the size of a/b is the length $\lceil \log_2(b+1) \rceil$ of the (shortest) binary notation for b.*

DEFINITION 2.8. *A domain Y with universe V is a* subdomain *of a domain X if V is a subset of (the universe of) X and $\mathbf{P}_X(V) > 0$ and $|x|_Y = |x|_X$, $\mathbf{P}_Y(x) = \mathbf{P}_X(x)/\mathbf{P}_X(V)$ for every $x \in V$. Y is also called the* restriction *$X|V$ of X to V.*

DEFINITION 2.9. *The direct product $X \times Y$ of domains X and Y is the domain of pairs (x, y), where $x \in X$ and $y \in Y$, such that $|(x, y)| = |x| + |y|$ and $\mathbf{P}(x, y) = \mathbf{P}_X(x) \times \mathbf{P}_Y(y)$.*

The direct product construction allows us to define powers $X^2, X^3, \ldots$ of a given domain X. Sometimes it is more natural to deal with subsets rather than sequences of elements of a given domain.

DEFINITION 2.10. *For each positive integer σ, $\mathrm{Set}_\sigma(X)$ is the domain of σ-element subsets S of a given domain X such that the size $|S| = \sum_{x\in S} |x|$ and the probability $\mathbf{P}(S)$ is proportional to the product $\prod_{x\in S} \mathbf{P}_X(x)$, i.e., $\mathbf{P}(S) \propto \prod_{x\in S} \mathbf{P}_X(x)$.*

There are various natural domains D of all finite nonempty subsets of a given domain X such that each $\mathrm{Set}_\sigma(X)$ is a subdomain of D.

DEFINITION 2.11. *$\mathrm{Set}(X)$ is the domain of nonempty subsets of X such that $|S| = \sum_{x\in S} |x|$ and $\mathbf{P}(S) \propto \prod_{x\in S} \mathbf{P}_X(x)$.*

We need to check that $\sum_S \mathbf{P}(S)$ converges. For each positive integer n, we have

$$1 = \left(\sum_{x\in X} \mathbf{P}(x)\right)^n = \sum_{x_1,\ldots,x_n\in X} \mathbf{P}(x_1)\cdots\mathbf{P}(x_n).$$

In the sum on the right, each $\mathbf{P}(S)$ occurs $n!$ times (and there are some additional terms with repeated x's). So $\sum_{S:\,|S|=n} \mathbf{P}(S) \le \frac{1}{n!}$.

Another possibility is to define a probability distribution on the collection of nonempty finite subsets of X that corresponds to the following experiment: first choose a positive integer n with respect to the default probability and then choose an n-element subset with respect to $\mathrm{Set}_n(X)$; call this alternative domain $\mathrm{Set}'(X)$.

In many respects, AP functions differ from polynomially bounded functions. Here is one illustration.

PROPOSITION 2.12. *For every countable family $\{f_1, f_2, \ldots\}$ of AP functions on BS there exists an AP function F on BS that is not majorized by any f_i.*

Proof. By induction on i, select elements x_i of BS such that

1. $|x_i| > |x_j|$ for all $j < i$, and
2. $2^{|x_i|} > f_i(x_i)$.

Such elements exist because each f_i is AP. Define

$$F(x) = \begin{cases} 2^{|x|} & \text{if } \exists i\,(x = x_i) \\ 0 & \text{otherwise.} \end{cases}$$

No f_i majorizes F because $F(x_i) > f_i(x_i)$. It remains to check that F is AP. We have

$$\sum_x \frac{1}{|x|} F(x)\mathbf{P}(x) \approx \sum_i \frac{1}{|x_i|} 2^{|x_i|} \frac{1}{|x_i|^2} 2^{-|x_i|} = \sum_i \frac{1}{|x_i|^3}$$

and the last sum is finite because all elements x_i are of different lengths. □

3. Domain reductions. In this section, we define and discuss many-one reductions, both deterministic and randomizing, between domains. These are the only sort of reductions used in this paper.

DEFINITION 3.1. *A function f from domain X to domain Y satisfies the* domination condition *if $\mathbf{P}_X[f^{-1}(fx)]/\mathbf{P}_Y(fx)$ is AP on X.*

COROLLARY 3.2. *An injective function f from domain X to domain Y satisfies the domination condition if and only if $\mathbf{P}_X(x)/\mathbf{P}_Y(fx)$ is AP on X.*

THEOREM 3.3 [BG1]. *Let f be an arbitrary function from a domain X to a domain Y. The following statements are equivalent:*

- *For every AP function T on Y, the composition $T \circ f$ is AP on X, and*
- *$|f(x)|_Y$ is AP on X and f satisfies the domination condition.*

This theorem, with T regarded as the running time of an algorithm on Y, suggests the following definition.

DEFINITION 3.4. *A* deterministic reduction *from a domain X to a domain Y is an AP time computable function f from X to Y such that $|f(x)|_Y$ is AP on X and f satisfies the domination condition.*

Such a reduction and an AP time algorithm on Y yield an AP time algorithm on X.

COROLLARY 3.5. *Deterministic reductions of domains compose. Therefore the relation of deterministic reducibility of domains is transitive.*

Also recall the two domains $\mathrm{Set}(X)$ and $\mathrm{Set}'(X)$ of finite nonempty subsets S of X defined above.

LEMMA 3.6. *The identity function reduces* $\mathrm{Set}(X)$ *to* $\mathrm{Set}'(X)$ *and* $\mathrm{Set}'(X)$ *to* $\mathrm{Set}(X)$.

Proof. The proof is obvious. □

LEMMA 3.7. *Let a function f reduce a domain X to a domain Y and suppose that f is one-to-one. Then, for each positive integer σ, the function*

$$F\{x_1, \ldots, x_\sigma\} = \{f(x_1), \ldots, f(x_\sigma)\}$$

reduces $\mathrm{Set}_\sigma(X)$ *to* $\mathrm{Set}_\sigma(Y)$.

Proof. Let σ be an arbitrary positive integer, $A = \mathrm{Set}_\sigma(X)$ and $B = \mathrm{Set}_\sigma(Y)$. It suffices to prove that F satisfies the domination condition. Since F is one-to-one, it suffices to prove that $\mathbf{P}_A(S)/\mathbf{P}_B(F(S))$ is AP on $\mathrm{Set}_\sigma(X)$.

Since f satisfies the domination condition, the ratio $\rho(x) = \mathbf{P}_X(x)/\mathbf{P}_Y(fx)$ is AP on X. Ignoring constant factors, we have

$$\frac{\mathbf{P}_A(S)}{\mathbf{P}_B(FS)} \approx \frac{\prod_{x\in S}\mathbf{P}_X(x)}{\prod_{x\in S}\mathbf{P}_Y(fx)} = \prod_{x\in S}\rho(x).$$

It remains to prove that the last product is AP on A.

Let δ witness that ρ is AP so that

$$\sum_{x\in X}\rho(x)^\delta \frac{1}{|x|}\mathbf{P}(x) < \infty.$$

It follows that

$$\sum_{x\in X}\rho(x)^{\delta/\sigma} \frac{1}{|x|^{1/\sigma}}\mathbf{P}(x) < \infty.$$

The terms where $\rho(x)\frac{1}{|x|} \le 1$ remain ≤ 1 and therefore sum up to at most 1, and the other terms become smaller. The number $\varepsilon = \delta/\sigma$ witnesses that $\prod_{x\in S}\rho(x)$ is AP on $\mathrm{Set}_\sigma(X)$. We have

$$\sum_S \left(\prod_{x\in S}\rho(x)\right)^\varepsilon \frac{1}{|S|}\prod_{x\in S}\mathbf{P}(x) \le \sum_S \prod_{x\in S}\frac{\rho(x)^\varepsilon \mathbf{P}(x)}{|x|^{1/\sigma}} \le \left[\sum_{z\in X}\frac{\rho(z)^\varepsilon \mathbf{P}(z)}{|z|^{1/\sigma}}\right]^\sigma < \infty.$$

The first of the three inequalities holds because the geometric mean $\prod_{x\in S}|x|^{1/\sigma}$ is bounded by the arithmetical mean of the same numbers $|x|$, which is bound by the size $|S|$. To prove the second inequality, note that every summand on the left-hand side is obtained when you multiply the σ copies of the infinite series. Concerning the third inequality, we have already checked that the infinite series in square brackets converges. It remains to apply Lemma 2.3. □

LEMMA 3.8. *Suppose that domains X and Y have the same universe and the same probability distribution. The identity function deterministically reduces X to Y if and only if the size function of Y is AP on X.*

Proof. Use Theorem 3.3. □

PROVISO 3.9. *Restrict attention to domains X such that the size function of X is AP on the domain X' obtained from X by redefining the size of a string as its length.*

COROLLARY 3.10. *A function f from a domain X to a domain Y reduces X to Y if and only if f is AP time computable and satisfies the domination condition.*

Proof. Let Y' be the domain obtained from Y by redefining the size as length. Since the length $\ell(f(x))$ of $f(x)$ is bounded by the time needed to compute $f(x)$, it is AP on X. Thus f reduces X to Y'. Now use Lemma 3.8 and the transitivity of the deterministic reducibility relation. □

Reductions are used in the usual way to define the notion of a complete problem in a complexity class, i.e., a problem in the class to which all problems in the class are reducible. Unfortunately, deterministic reductions are too weak to yield a good notion of completeness; see [Gu1] where it is shown that complete problems in this sense must (under the reasonable assumption that nondeterministic exponential time differs from deterministic exponential time) have special probability distributions (nonflat, in the terminology of [Gu1]). Therefore, we use the larger class, suggested in [VL], of randomizing reductions, i.e., we allow the computation of the reducing function to flip coins. To introduce randomizing reductions, we need some auxiliary notions.

Terminology and notation. A set S of binary strings satisfies the *prefix condition* if no string in S is a prefix of a different string in S. If Δ is a subset of the cartesian product $U \times V$ of sets U and V then, for each $x \in U$, $\Delta(x) = \{y : (x, y) \in \Delta\}$.

The notion of dilation was introduced in [Gu1] and used in [BG1], [BG2]. The idea is to combine the probability distribution on instances and the probability distribution of coin flips into one probability distribution. In the following definition, think of Δ as the set of pairs (x, s) where x is an input to a randomizing algorithm and s is a sequence of coin flips just sufficient to make that algorithm, with input x, produce an output.

DEFINITION 3.11 (cf. [Gu1], [BG1]). *A* dilation *of a domain X is a domain Δ such that*

- *the universe of Δ is a subset of $X \times BS$ such that, for every $x \in X$ of nonzero probability, $\Delta(x)$ is nonempty and satisfies the prefix condition*;
- *the size function $|(x, s)| = |x|$; and*
- *the probability distribution $\mathbf{P}(x, s) = \mathbf{P}(x)2^{-|s|}/\sum_{t\in\Delta(x)} 2^{-|t|}$.*

DEFINITION 3.12. *Let Δ be a dilation of a domain X. Then*

$$\text{Density}_\Delta(x) = \sum_{s\in\Delta(x)} 2^{-|s|} \quad \textit{and} \quad \text{Rarity}_\Delta(x) = \frac{1}{\text{Density}_\Delta(x)}.$$

Further, Δ is nonrare *if the rarity function $\text{Rarity}_\Delta(x)$ is AP on X. Δ is* almost total *if $\text{Rarity}_\Delta(x) = 1$ for every x of nonzero probability; in terms of coin flips, that means that, if we repeatedly flip a fair coin to produce a string of 0's and 1's, then, with probability 1, we will eventually obtain a string in $\Delta(x)$. Finally, Δ is* trivial *if, for every $x \in X$ of nonzero probability, $\Delta(x)$ contains the empty string (and therefore contains no other string).*

Theorem 3.3 generalizes to randomizing reductions.

THEOREM 3.13 [BG2]. *Suppose that Γ is a nonrare dilation of a domain X and f is a function from Γ to Y. Then the following statements are equivalent:*

- *$|f(x)|_Y$ is AP on Γ and f satisfies the domination condition;*
- *for every nonrare dilation Δ of Y and every AP function T on Δ, the composition $T(f(x,s),t)$ is AP on the dilation $\Gamma * \Delta$ of A that comprises pairs (x, st) such that $(x,s) \in \Gamma$ and $(f(x,s),t) \in \Delta$.*

DEFINITION 3.14. *A* (randomizing) reduction *of a domain X to a domain Y consists of a nonrare dilation Γ of X and a deterministic reduction f of Γ to Y.*

Randomizing reductions of domains compose in the following sense. If Γ and Δ are nonrare dilations of X and Y, respectively, and and if $f : \Gamma \longrightarrow Y$ and $g : \Delta \longrightarrow Z$ are randomizing reductions of X to Y and of Y to Z, then there is a composite reduction $g \circ f : \Gamma * \Delta \longrightarrow Z$ of X to Z. Here $\Gamma * \Delta$ is as in Theorem 3.13 and $g \circ f$ is defined by $(g \circ f)(x, st) = g(f(x,s),t)$ whenever $(x,s) \in \Gamma$ and $(f(x,s),t) \in \Delta$. (Although this composition is not the ordinary composition of functions, it does yield a category of domains and random functions.)

COROLLARY 3.15. *The relation of (randomized) reducibility is transitive.*

DEFINITION 3.16. *Let Σ be an alphabet. A* randomizing algorithm *on Σ^* is an algorithm A on $\Sigma^* \times BS$, but the two input strings, a string x over Σ and a binary string s, play different roles. The string x is viewed as the input, and the string s is viewed as a sequence of coin flips. It is supposed that A does not flip a coin unless the computation requires another random bit.*

DEFINITION 3.17. *A dilation Δ of a domain X is* (AP time) certifiable *if there exists a randomizing algorithm A on Σ_X^* such that*

- *for every $x \in X$ of nonzero probability and every binary string s, A outputs YES on input (x,s) if and only if $(x,s) \in \Delta$, and*
- *the computation time of A is AP on Δ.*

The need for certifiable reductions arises when dealing with decision problems. We consider algorithms that, given an input x of nonzero probability, produce a correct output on any random string $s \in \Delta(x)$ but may produce an incorrect output on $s \notin \Delta(x)$. When the correctness of the output cannot be verified efficiently, the certifiability of Δ will be needed to justify believing the output.

DEFINITION 3.18. *A reduction (Γ, f) of a domain X to a domain Y is* certifiable *if Γ is certifiable.*

LEMMA 3.19. *Certifiable reductions of domains compose.*

Proof. Chase the definitions and apply Theorem 3.3. □

As an example of how much easier it may be to deal with randomizing reductions, consider the problem of reducing BS to FRACTION. To avoid trivial solutions, like constant functions, let us require that different elements of BS are taken to different elements of FRACTION. The problem is easily solved with the help of randomization.

LEMMA 3.20. *There exists a randomized reduction (Γ, f) from BS to FRACTION such that $f(x_1,s_1) \neq f(x_2,s_2)$ whenever $x_1 \neq x_2$.*

Proof. $\Gamma(x)$ comprises all binary strings s of length $|x|$ such that the numbers (represented by binary strings) s and $1x$ are relatively prime. Since the chance that a random s is relatively prime to $1x$ is sufficiently large [HW], Γ is nonrare. $f(x,s) = s/1x$. □

4. Search problems.

DEFINITION 4.1. *A* (randomized) search problem SP(X, W) *is given by a domain X (of* instances*) and a PTime computable relation $W(x,w)$ (the* witness relation*) between elements of X and arbitrary strings in a fixed alphabet. The problem is: Given an instance x with $W(x) \neq \emptyset$, find an element of $W(x)$ (a* witness *for x).*

DEFINITION 4.2. SP(X, W) *is* AP time solvable *if there exist a nonrare dilation* Γ *of* $X|\{x : W(x) \neq \emptyset\}$ *and an AP time algorithm* M *on* Γ *that, given any* $(x, s) \in \Gamma$ *with* $\mathbf{P}_X(x) > 0$, *finds a witness for* x. *Such a pair* (Γ, M) *is called an* AP time solution *for* SP(X, W). *A solution* (Γ, M) *is* almost total *if* Γ *is so.*

This notion of AP solvability may seem weaker than it is.

THEOREM 4.3 [BG2]. *Every AP time solvable search problem* SP(X, W) *has an almost total solution.*

DEFINITION 4.4. *A* (randomizing) reduction *of* SP(X, U) *to* SP(Y, V) *consists of*

Dilation: *A nonrare dilation* Γ *of* X,

Instance transformer: *A deterministic reduction* f *of* Γ *to* Y, *and*

Witness transformer: *A PTime computable function* $g((x, s), v)$ *such that if* $s \in \Gamma(x)$ *and* $v \in V(f(x, s))$ *then* $g((x, s), v) \in U(x)$.

THEOREM 4.5 [BG2]. *The reducibility relation on search problems is transitive, and a problem* SP(X, U) *is solvable in AP time if it is reducible to some problem* SP(Y, V) *which is solvable in AP time.*

The notion of reduction allows us to define complete problems in the usual way. SP(X, W) is *complete* for a class $\mathcal{C}$ of search problems if it is in $\mathcal{C}$ and every problem in $\mathcal{C}$ reduces to it.

5. Decision problems.

DEFINITION 5.1. *A* (randomized) decision problem DP(X, P) *is given by a domain* X *of* instances *and a subset* P *of* X. *Instances in* P *are called* positive, *and instances in* $X - P$ *are called* negative. *The problem is: Given an instance* $x \in X$, *decide whether* x *is positive or negative.*

DEFINITION 5.2. *DP*(X, P) *is* AP time solvable (*or* AP time decidable) *if there exist a nonrare certifiable dilation* Γ *of* X *and an AP time algorithm* M *on* Γ *that, given any element* $(x, s) \in \Gamma$ *with* $\mathbf{P}_X(x) > 0$, *decides whether* x *is positive or negative. The pair* (Γ, M) *is an* AP time solution *for* DP(X, P). *A solution* (Γ, M) *for* DP(X, P) *is* almost total *if* Γ *is so.*

Note that certifiability is required, as we cannot check whether the output of M (yes or no) is correct. Contrast this with search problems where the assumed computability of the witness relation lets us check whether the output (an alleged witness) is indeed a witness and certifiability is therefore not required.

Again, the notion of AP time solution may seem weaker than it is.

THEOREM 5.3. *If* DP(X, P) *is AP time solvable then it has an almost total AP time solution.*

Proof. Let (Γ, M) be an AP time solution for a decision problem DP(X, P) and let A be a certifying algorithm for Γ. For each $s \in \Gamma(x)$, let s' be the computation of A on (x, s) and s'' be the computation of M on (x, s). Define

$$W = \{(x, (s, s', s'')) : (x, s) \in \Gamma\}.$$

Obviously, the relation W is PTime computable. (The intended algorithm for computing W uses A and M; we need not check $(x, s) \in \Gamma$ because this follows from $A(x, s)$ providing s'.) The dilation Γ and a combination of the algorithms A and M give an AP time solution for the search problem SP(X, W). By Theorem 4.3, this search problem has an almost total AP time solution (Δ, N). For each $(x, t) \in \Delta$ with $\mathbf{P}_X(x) > 0$, N outputs a triple $(s, s', s'') \in W$. By the definition of W, $s = s(t) \in \Gamma(x)$ and therefore $s(t)''$ is a computation of M deciding whether x is positive or negative.

The dilation Δ is certifiable. Given an instance x of nonzero probability and an arbitrary string t, the desired certifying algorithm A' runs N on (x, t). If a prefix t_0 of t belongs to $\Delta(x)$, N will produce an output on (x, t_0) and A' will output "yes" in the case where $t_0 = t$ or "no" in the case where t_0 is a proper prefix of t. Suppose that no prefix of t belongs to $\Delta(x)$.

Since Δ is almost total, both $t0$ and $t1$ are prefixes of strings in Δ. The computation of N on (x, t) will stop without producing any output, waiting for another random bit; in such a case A' will output "no."

Let N' be the modification of N that, given $(x, t) \in \Delta$ with $\mathbf{P}_X(x) > 0$, outputs only the result (yes or no) of the computation $s(t)''$. The pair (Δ, N') constitutes an almost total AP time solution for DP(X, P). □

Note the role of the certifying algorithm A. The certifiability of dilation was unnecessary in the case of search problems, but in the case of decision problems it plays an important role.

DEFINITION 5.4. *A* (randomizing) many-one reduction *of* DP(X, P) *to* DP(Y, Q) *comprises*

- *a nonrare certifiable dilation* Γ *of* X, *and*
- *a deterministic reduction* f *from* Γ *to* Y (*the* instance transformer) *satisfying the following* correctness property: *For all* $(x, s) \in \Gamma$,

$$f(x, s) \in Q \iff x \in P.$$

THEOREM 5.5. *The many-one reducibility relation on decision problems is transitive, and a problem* DP(X, P) *is AP time decidable if it reduces to some problem* DP(Y, Q) *that is AP time decidable.*

Proof. Use the fact, established in §3, that randomizing domain reductions compose. □

DEFINITION 5.6. *A many-one reduction* (Γ, f) *of* DP(X, P) *to* DP(Y, Q) *is* deterministic *if* Γ *is trivial. In this case, the reduction* (Γ, f) *is specified only by the instance transformer* f; Γ *may be identified with* X.

LEMMA 5.7. *The identity function deterministically reduces any decision problem* Π *to the decision problem* Π' *obtained from* Π *by redefining the size of a string as its length.*

Proof. Use Corollary 3.10. □

A decision problem is *hard* for a class $\mathcal{C}$ of decision problems if every problem in $\mathcal{C}$ reduces to it. A decision problem is *complete* for $\mathcal{C}$ if it belongs to $\mathcal{C}$ and is hard for $\mathcal{C}$.

RNP is the class of decision problems with PTime computable probability distributions. PTime computable distributions are defined in [Le] and analyzed in [Gu1].

6. Positive matrices. We now turn from the general theory of randomizing algorithms and reductions to the specific problem, Matrix Transformation, whose completeness for RNP we prove in § 10. We begin with information about unimodular matrices.

Call a unimodular matrix (i.e., an element of $\mathrm{SL}_2(Z)$, a two-by-two matrix with determinant 1) *positive* if it has no negative entries. Positive matrices form a monoid $PM = \mathrm{SL}_2(N)$. In this section, a column is a column of two relatively prime nonnegative integers; for notational simplicity, we view a positive matrix as the pair of its columns. If u is a column, let u_1 be the upper and u_2 the lower components of u. Partially order columns componentwise: $u \leq v$ if $u_1 \leq v_1$ and $u_2 \leq v_2$, and $u < v$ if $u \leq v$ and either $u_1 < v_1$ or $u_2 < v_2$. Define $\max(X)$ to be the maximal entry of a positive matrix X. In this section,

$$A_0 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \quad \text{and} \quad B_0 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}.$$

LEMMA 6.1.

1. $(u, v) \times A_0 = (u + v, v)$, *and* $(u, v) \times B_0 = (u, u + v)$.
2. *If* A_0 *is a right divisor of a positive matrix* (u, v) *in* PM *then* $u > v$, *and if* B_0 *is a right divisor of* (u, v) *in* PM *then* $u < v$.
3. *If the maximal entry* m *of a matrix* (u, v) *appears in two or more places then* $m = 1$.

Proof. Part (1) is obvious, and (2) follows from (1).

(3) If m occurs twice in the same row or the same column, then it divides the determinant 1 and therefore $m = 1$. So assume this does not happen. If $v_1 = u_2 = m$ then the determinant cannot be positive. If $u_1 = v_2 = m$ then $1 = u_1v_2 - v_1u_2 \geq m^2 - (m-1)^2 = 2m - 1$ and therefore $m = 1$. □

The second statement of Lemma 6.1 implies that the monoid generated by the matrices A_0 and B_0 is free. This fact is noted in [Ei, Chap. VI, §12]. The following theorem should be known too, but we do not have an appropriate reference.

THEOREM 6.2. *The monoid PM is isomorphic to the monoid BS of binary strings. The two indecomposable nonunit elements are the matrices A_0 and B_0.*

Proof. Since A_0 and B_0 generate a free monoid, it suffices to prove that every nonunit positive matrix (u, v) is a product of matrices A_0 and B_0. Define weight$(u) = u_1 + u_2$ and weight$(u, v) =$ weight$(u)+$ weight(v). The proof is an induction on $s =$ weight(u, v). Since the entries of the main diagonal are not zero, $s \geq 2$.

The case $s \leq 3$ is easy: A_0 and B_0 are the only nonunit matrices of weight ≤ 3. Suppose that $s > 3$. Then $m = \max(u, v) > 1$. Exploiting the symmetry, we may suppose that m appears in u. If $u_1 = m$ then $1/m = (u_1v_2 - v_1u_2)/m > v_2 - u_2$ and therefore $u_2 \geq v_2$. Similarly, if $u_2 = m$ then $u_1 \geq v_1$. Thus, the column $u - v$ has nonnegative entries. The determinant of $(u - v, v)$ equals 1 and therefore $(u - v, v)$ is an element of $\mathrm{SL}_2(N)$. By the induction hypothesis, $(u - v, v)$ is a product of matrices A_0 and B_0. By Lemma 6.1(1), $(u, v) = (u - v, v) \times A_0$. □

COROLLARY 6.3. *If a positive matrix (u, v) is not the unit matrix then one of the two columns is greater than the other.*

Proof. The fact has been established in the proof of Theorem 6.2. □

Call the greater column of a nonunit positive matrix *major*; in the case of the unit matrix, call either column *major*. The other column of the matrix will be called *minor*.

LEMMA 6.4. *The major column and one bit indicating whether it is the first or the second column uniquely define the minor column.*

Proof. Without loss of generality, the given matrix (u, v) is not the unit matrix. It follows that both components of the major column are positive. By virtue of symmetry, suppose that u is the major column. We show that the minor column v is the least column such that $u_1v_2 - u_2v_1 = 1$. Let w be any column such that $u_1w_2 - u_2w_1 = 1$. Then $u_1(w_2 - v_2) = u_2(w_1 - v_1) = u_1u_2k$ for some k because u_1 and u_2 are relatively prime; hence $w_1 = v_1 + ku_1$ and $w_2 = v_2 + ku_2$. If $k < 0$ then either w_1 or w_2 is negative. Hence $k \geq 0$ and therefore $w_1 \geq v_1$, $w_2 \geq v_2$. □

LEMMA 6.5. *There exists a PTime algorithm that, given a column u, computes the minor column of the unique positive unimodular matrix with the first and major column u.*

Proof. Use the extended Euclid's algorithm [Kn1]. □

Remark 1. Instead of columns, we could use rows above in this section. This would cause some insignificant changes in Lemma 6.1 (for example, the first statement would say that $A_0 \times (u, v) = (u, u + v)$, and $B_0 \times (u, v) = (u + v, v)$ where u is the upper row of the given matrix and v is the lower row), but Corollary 6.3 and Lemma 6.4 remain true.

Let $\ell(n)$ be the length of the binary notation for n.

DEFINITION 6.6. *We define a domain structure on the monoid PM. It is the uniform domain with the size function $|X| = \ell(\max(X))$. Thus, PM is the monoid and domain of positive matrices.*

Strictly speaking, the elements of a domain should be strings. For this purpose, we may regard a matrix as a list of its entries in binary notation. Then Proviso 3.9 is satisfied.

LEMMA 6.7. *The relative probability* $\mathbf{P}_{\mathrm{PM}}[X \mid |X| = l] = \Theta(2^{-2l})$.

Proof. Let $g(l) \approx f(l)$ mean that $g(l) = \Theta(f(l))$, i.e., that there exist positive constants c, c', and l_0 such that $cf(l) \leq g(l) \leq c'f(l)$ for all $l \geq l_0$ [Kn2]. It suffices to prove that the number $N(l)$ of positive matrices of size l is $\Theta(2^{2l})$. Recall that $\phi(m)$ is the number of positive integers $n \leq m$ that are prime to m, and that $\Phi(m) = \phi(1) + \cdots + \phi(m) = 3m^2/\pi^2 + O(m \cdot \log m)$ [HW, Thm. 330]. Thus,

$$N(l) \approx \sum_{\ell(m)=l} \phi(m) \approx \Phi(2^l - 1) - \Phi(2^{l-1}) = \Theta(2^{2l}). \qquad \square$$

By Theorem 6.2, PM is isomorphic to BS as a monoid. There are exactly two isomorphisms of PM onto BS. One of them takes A_0 to 0 and B_0 to 1 while the other one takes A_0 to 1 and B_0 to 0. Let I be the isomorphism that takes A_0 to 0, and let J be the corresponding isomorphism I^{-1} from binary strings to PM. It is easy to check by induction on the length of the given string x that if x starts with a zero (respectively, one) then the lower (respectively, upper) row of $J(x)$ is major (see "transposed" Lemma 6.1 in Remark 6). Notice that the size of a matrix X may be quite different from the length of the corresponding string $I(X)$. It is easy to see that the isomorphism I is not computable in polynomial time: A matrix $A_0^n = \binom{1}{n}\ \binom{0}{1}$ is of size $\ell(n)$ whereas the string $0^n = I(A_0^n)$ is of length n.

LEMMA 6.8. *I is AP time computable.*

Proof. The following recursive algorithm computes $I(X)$. If X is the unit matrix then $I(X)$ is the empty string. Suppose that $X = (u, v)$ differs from the unit matrix. If u is the major column, $w = u - v$ and $z = I(w, v)$ then $I(X) = z0$, and if v is the major column, $w = v - u$ and $z = I(u, w)$ then $I(X) = z1$. The computation time of that algorithm is essentially proportional to $|I(X)|$, which is AP by Lemma 6.12. $\square$

The isomorphism J is PTime computable but $\mathbf{P}_{\mathrm{PM}}$ does not dominate $\mathbf{P}_{\mathrm{BS}}$ with respect to J and thus J fails to reduce BS to PM.

THEOREM 6.9. *$\mathbf{P}_{\mathrm{PM}}$ does not dominate $\mathbf{P}_{\mathrm{BS}}$ with respect to J.*

Proof. By contradiction, suppose that the function $g(x) = \mathbf{P}_{\mathrm{BS}}[x]/\mathbf{P}_{\mathrm{PM}}[Jx]$ is polynomial on average with respect to $\mathbf{P}_{\mathrm{BS}}$ and fix $\varepsilon > 0$ to witness that fact. Thus,

$$\infty > \sum_n \sum_{x:\,|x|=n} \frac{1}{n}(g(x))^{\varepsilon} \frac{1}{n(n+1)} 2^{-n},$$

so, as a function of n, the conditional expectation of $(g(x))^{\varepsilon}$ for strings of length n, $\sum_{x:\,|x|=n}(g(x))^{\varepsilon}2^{-n}$ is $o(n^3)$. We obtain the desired contradiction by proving that this expectation is not bounded by any polynomial of n.

Let $l = |Jx|$. By Lemma 6.7,

$$g(x) = \Theta(l^2 \cdot 2^{2l} \cdot n^{-2} \cdot 2^{-n}).$$

Let $s(x)$ be the sum of the entries of the major row of Jx. Clearly, $s(x) = \Theta(2^l)$. Hence it suffices to prove that the expectation $E_n = \sum_x [(s(x)^2/2^n)^{\varepsilon} \cdot 2^{-n}]$ of the function $[s(x)^2/2^n]^{\varepsilon}$ is not bounded by a polynomial of n. (The factor n^{-2} in $g(x)$ will not matter as it is the reciprocal of a polynomial, and l^2 will not matter as it is ≥ 1.) We may restrict our attention to $\varepsilon < 1/2$. Let y range over strings of length $n - 1$. $\square$

LEMMA 6.10. *There exists some $\alpha > 1$ such that every*

$$A(y) = (1/2)[s(y0)^{2\varepsilon} + s(y1)^{2\varepsilon}]/s(y)^{2\varepsilon} \geq \alpha 2^{\varepsilon}$$

Proof. Let $a > b$ be the two entries of the major row of $J(y)$, and $\gamma = b/a$. Then

$$A(y) = (1/2)[(2a+b)^{2\varepsilon} + (a+2b)^{2\varepsilon}]/(a+b)^{2\varepsilon} = (1/2)[(2+\gamma)^{2\varepsilon} + (1+2\gamma)^{2\varepsilon}]/(1+\gamma)^{2\varepsilon}.$$

Let $\delta = 1/(1+\gamma)$. Then $A(y) = (1/2)[(1+\delta)^{2\varepsilon} + (2-\delta)^{2\varepsilon}]$.

Consider the function $f(t) = t^{2\varepsilon}$ of a real variable t. The graph of f is concave because $f''(t) = 2\varepsilon(2\varepsilon - 1)t^{2\varepsilon-2} < 0$. Since $0 < \delta < 1$, the chord C between the points $(1+\delta, f(1+\delta))$ and $(2-\delta, f(2-\delta))$ lies strictly above the chord C_0 between the points $(1, f(1))$ and $(2, f(2))$. Further, the center of the interval $[1+\delta, 2-\delta]$ coincides with the center 1.5 of the interval $[1, 2]$, and therefore the center $(1.5, A(y))$ of C lies directly above the center $(1.5, [1+2^{2\varepsilon}]/2)$ of C_0. The arithmetical mean $[1+2^{2\varepsilon}]/2$ of numbers 1 and $2^{2\varepsilon}$ exceeds their geometrical mean 2^{ε}. Thus, $A(y) > [1+2^{2\varepsilon}]/2 > 2^{\varepsilon}$. The desired $\alpha = (1/2)[1+2^{2\varepsilon}]/2^{\varepsilon}$. □

We continue to prove Theorem 6.9. Let $A(y)$ and α be as in Lemma 6.10, and let x range over strings of length n and y over strings of length $n-1$. We have

$$E_n = \sum_x [(s(x)^2/2^n)^{\varepsilon} \cdot 2^{-n}] = 2^{-n\varepsilon} \cdot \sum_x [s(x)^{2\varepsilon} \cdot 2^{-n}] \geq 2^{-n\varepsilon} \cdot \sum_y [s(y)^{2\varepsilon} \cdot A(y) \cdot 2^{-(n-1)}].$$

By Lemma 6.10,

$$E_n \geq 2^{-n\varepsilon} \cdot \sum_y [s(y)^{2\varepsilon} \cdot \alpha 2^{\varepsilon} \cdot 2^{-(n-1)}] = \alpha \sum_y [(s(y)^2/2^{n-1})^{\varepsilon} \cdot 2^{-(n-1)}] = \alpha E_{n-1}.$$

It follows that E_n is $\Omega(\alpha^n)$ and therefore is not bounded by a polynomial of n. □

Recall the notion of a (simple) continued fraction [HW]. Here is an example:

$$\frac{81}{17} = 4 + \frac{13}{17} = 4 + \frac{1}{\left(\frac{17}{13}\right)} = 4 + \frac{1}{1+\frac{4}{13}} = 4 + \frac{1}{1+\frac{1}{\left(\frac{13}{4}\right)}} = 4 + \frac{1}{1+\frac{1}{3+\frac{1}{4}}} = [4, 1, 3, 4].$$

Similarly, every positive rational number r can be uniquely represented by a continued fraction $[a_l, \ldots, a_0]$ where a_l is a nonnegative integer, and each a_i with $0 < i < l$ is a positive integer, and a_0 is an integer ≥ 2 unless $l = 0$; the integers $a_i \geq 0$ are called partial quotients.

LEMMA 6.11. *Suppose that x is a nonempty binary string and let $m \leq n$ be the two entries of the major row of $J(x)$. Then $|x|$ equals the sum $s(n, m)$ of the partial quotients in the continued fraction for n/m.*

Proof. If $|x| = 1$ then $m = n = 1$ and $s(n, m) = 1 = |x|$. Suppose that $|x| > 1$. By virtue of symmetry, we may suppose that $x = y0$; the other case is similar. Let (i, j) be the major row of $J(y)$. By Lemma 6.1, the major row (n, m) of $J(x)$ is $(i+j, j)$. It suffices to prove that if $i \leq j$ then $s(n, m) = s(j, i) + 1$, and if $i \geq j$ then $s(n, m) = s(i, j) + 1$. Since $J(y)$ is not the unit matrix, neither i nor j is zero. In any case,

$$\frac{n}{m} = \frac{i+j}{i} = 1 + \frac{j}{i}.$$

If $i \leq j$ and $\frac{j}{i} = [a_l, \ldots, a_0]$ then $\frac{n}{m} = [a_l + 1, \ldots, a_0]$, so $s(n, m) = s(j, i) + 1$.

If $j \leq i$ and $\frac{i}{j} = [a_l, \ldots, a_0]$ then

$$\frac{n}{m} = 1 + \frac{1}{\left(\frac{1}{j}\right)} = [1, a_l, \ldots, a_0],$$

so $s(n, m) = s(i, j) + 1$. □

LEMMA 6.12. *$|I(X)|$ is AP on PM.*

Proof. Let $s(n, m)$ be as in Lemma 6.11. We use the following strong result of Yao and Knuth [YK]: $\sum_{m=1}^{m=n} s(n, m) = (6n/\pi^2)(\ln n)^2 + O(n(\log n)(\log\log n)^2) = \Theta(n(\log n)^2)$. Let X be a matrix of size $l > 0$, and let $a(X) < b(X)$ form the major row of the matrix X. Then $\sum_{X:\, b(X)=n} s(b(X), a(X)) = \Theta(n(\log n)^2)$. By Lemma 6.11 $\sum_{b(X)=n} |I(X)| = \Theta(n(\log n)^2)$ and therefore $\sum_{X:\, |X|=l} |I(X)| = \Omega(2^l \cdot l^2 2^l)$. Now use Lemma 6.7 to check that the expectation of $|I(X)|$ with respect to the conditional probability $\mathbf{P}_{\mathrm{PM}}[X \mid |X| = l]$ is bounded by a polynomial of l. It follows (see Lemma 2.3) that $I(X)$ is AP on PM. □

7. Positive matrix correspondence problem.

DEFINITION 7.1. *Let T be a nondeterministic Turing machine with binary input alphabet. The* bounded halting problem BH(T) *is the randomized decision problem with domain* BS × PI *such that an instance (x, n) is positive if and only if T has a halting computation of length $\leq n$ on x. Call an instance (x, n) of* BH(T) robust *if either T has a halting computation of length $\leq n$ on x or else T has no halting computation on x at all.* RBH(T) *is the restriction of* BH(T) *to robust instances.*

DEFINITION 7.2. *WBS is the domain of binary strings where $|x|$ is the length of x and $\mathbf{P}(x) = \mathbf{P}_{\mathrm{PM}}(Jx)$. Let T be a nondeterministic Turing machine with binary input alphabet. The* weird halting problem WH(T) *and its robust version* RWH(T) *are similar to* BH(T) *and* RBH(T) *except the domain is* WBS × PI *rather than* BS × PI.

LEMMA 7.3. *For a certain U,* RWH(U) *is hard for RNP.*

Proof. Some RBH(T) are RNP complete, by Corollary 1 of Theorem 1 in [Gu1]. (Actually, a slightly different version of bounded halting problems was considered in [Gu1]. It was supposed there that $n > |x|$ and $\mathbf{P}(x, n) \propto n^{-3}2^{|x|}$. However the same proof works. Also, the identity function deterministically reduces that older version of every RBH(T) to the new one.) Thus it suffices to reduce an arbitrary RBH(T) to an appropriate RWH(U). We will do just that.

One might be tempted to take $U = T$ and to use the identity mapping as a reduction. By Theorem 6.9, the identity function fails to do the job.

For every binary string s, let $N(s)$ be the integer with binary representation $1s$. If $N(s) = k$, let $S(k) = s$. Given a binary string y, the desired U computes $x = S(\max(J(y)))$, turns itself into T and then runs on input x. We construct a reduction (Γ, f) from RBH(T) to RWH(U). Here Γ is a dilation of BS × PI, and f is a function from Γ to WBS × PI.

Define $\Gamma(x, n)$ to comprise binary strings s of length $\geq |x| - 1$ such that $N(s)$ is prime to and less than $N(x)$. It is obvious that the dilation Γ is certifiable. By Theorem 328 in [HW], the number $\phi(k)$ of positive integers prime to and less than k is $\Omega(k/\log\log k)$. Note that, if j is relatively prime to k, then so is $k - j$. Thus, half of the integers counted by $\phi(k)$ are $\geq \frac{k}{2}$, and so the cardinality of $\Gamma(x, n)$ is $\geq \frac{1}{2}\phi(N(x))$. It follows that Γ is not rare:

$$\mathrm{Density}_\Gamma(x) = \sum_{s\in\Gamma(x)} 2^{-|s|} \geq \sum_{s\in\Gamma(x)} 2^{-|x|} = \Omega\left(\frac{1}{\log\log N(x)}\right) = \Omega\left(\frac{1}{\log|x|}\right).$$

By Lemma 6.4, for each $(x, n, s) \in \Gamma$, there exists a unique positive unimodular matrix $M(x, s)$ with first and major column $(N(x), N(s))$. View M as a function on Γ; we check that it reduces Γ to PM. By Lemma 6.5, the function M is PTime computable. To check the domination property, use Lemma 6.7 and the fact that M is injective.

The desired f is given by $f((x, n), s) = (y, t(x, s) + n)$ where $y = I(M(x, s))$ and $t(x, s)$ is the time that U needs to convert y into x. First, we check that f takes robust instances of BH(T) to robust instances of WH(U) and has the correctness property. By the definition of U, it halts on y if and only if T halts on x. Moreover, if T halts within $\leq n$ steps on x then U

halts within $t(x, s) + n$ steps on y. Now suppose that U halts on y. Then T halts on x. Since (x, n) is robust, T halts within n steps on x. Hence U halts within $t(x, s) + n$ steps on y.

Next, we check that f is AP time computable on Γ. We have to prove that y and $t(x, s)$ are computable in AP time relative to Γ. In the case of y this follows from the definition $y = I(M(x, s))$, the fact that M reduces Γ to PM, Lemmas 6.8 and 6.12 and Theorem 3.3. Now consider $t(x, s)$. It is the time that U needs to compute $x = S(\max(J(y)))$ from y, which is bounded by a polynomial of $|y|$ because J, max, and S are all PTime computable. So, by Lemma 2.3, $t(x, s)$ is AP on Γ. In addition, it is computable in AP time relative to Γ, because it can be computed by first computing y (which we already saw takes AP time) and then computing x from y while running a clock (which takes time essentially $t(x, s)$).

Finally, we check that f has the domination property. Notice that f is one-to-one, so we can use Corollary 3.2. In addition, the probability functions for WBS $\times$ PI and its restriction to robust instances differ by a constant factor, so we can compute with the former instead of the latter. We have

$$\frac{\mathbf{P}_\Gamma((x, n), s)}{\mathbf{P}_{\mathrm{WBS}\times\mathrm{PI}}(y, t(x, s) + n))} = \frac{\mathbf{P}_\Gamma((x, n), s)}{\mathbf{P}_{\mathrm{WBS}}(y)} \times (t(x, s) + n)(t(x, s) + n + 1).$$

The fraction on the right is AP on Γ because $\mathbf{P}_{\mathrm{WBS}}(y) = \mathbf{P}_{\mathrm{PM}}(M(x, s))$ and M has the domination property. Now use Lemma 2.3 and the fact that $t(x, s)$ is AP on Γ. □

The direct product PM $\times$ PM is a domain and monoid of positive matrix pairs; the multiplication of matrix pairs is componentwise: $(X_1, Y_1) \times (X_2, Y_2) = (X_1X_2, Y_1Y_2)$. If S is a set of positive matrix pairs, let S^n comprise products $P_1 \times \ldots \times P_m$ where $m \leq n$ and each $P_i \in S$. In the following definition, PMC stands for positive matrix correspondence.

DEFINITION 7.4. *For each positive integer* σ, PMC(σ) *is the decision problem with domain*

$$\mathrm{PM} \times \mathrm{Set}_\sigma(\mathrm{PM} \times \mathrm{PM}) \times \mathrm{PI},$$

where an instance (A, S, n) *is positive if and only if there exists a pair* (X, Y) *in* S^n *such that* $AX = Y$. *An instance* (A, S, n) *of* PMC(σ) *is* robust *if either* $AX = Y$ *for some pair* (X, Y) *in* S^n *or else the whole submonoid of* PM $\times$ PM *generated by* S *has no pair* (X, Y) *with* $AX = Y$. RPMC(σ) *is the restriction of* PMC(σ) *to robust instances* (A, S, n).

THEOREM 7.5. *Some* RPMC(σ) *is hard for RNP.*

Remark 2. Replacing PM with BS in the definition of RPMC(σ) gives a variant RPCP(σ) of the Post Correspondence Problem; this variant has been proved RNP complete for not too small σ in [Gu1]. (Actually, RPCP(σ) is slightly different from the version in [Gu1] but the difference is immaterial. For readers with §5.1 of [Gu1] before them, we indicate the changes needed in the completeness proof to cover our variant. Remove the clause L0 from the definition of $L(w)$ to obtain a reduced set $L'(w)$ of pairs. Instead of the instance $(L(w), p(m))$, use the instance $(ws_0, L'(w), p(m))$.) If we ignore probabilities and deal with decision problems only then the isomorphism J of §3 gives rise to a polynomial time reduction of RPCP(σ) to PMC(σ). Unfortunately, this reduction fails to have the domination property and it is difficult to alter in any way: The correctness property of the reduction is too closely related to fact that J is an isomorphism. Theorem 7.5 is not proved by a reduction from RPCP(σ), but the proof of completeness of RPCP(σ) is used in an essential way.

Proof of Theorem 7.5. Fix a Turing machine U witnessing Lemma 7.3. We will reduce RWH(U) to RPMC(σ) for appropriate σ. The variant RPCP(σ) of the Post Correspondence Problem was defined in a remark above. According [Gu1, §5] (with changes indicated in the remark), there exists a PTime reduction

$$F(x, n) = (xx', K(x), p(n))$$

of RBH(U) to some RPCP(σ) such that $|x'| = O(\log|xx'|)$ and $|K(x)| = O(\log|xx'|)$. (Concerning the size bounds, see in particular Lemma 5.1 in [Gu1].) Extend the isomorphism J to sequences of pairs of binary strings. The function

$$G(x, n) = (J(xx'), J(K(x)), p(n))$$

is the desired reduction of RWH(U) to RPMC(σ). Clearly, G is AP time computable and has the correctness property. Ignoring factors bounded by a polynomial of $|x| + n$ from above and by an inverse polynomial of $|x| + n$ from below, we have

$$\mathbf{P}_{\mathrm{RPMC}(\sigma)}[G(x, n)] \geq \mathbf{P}_{\mathrm{PM}}[J(x)] = \mathbf{P}_{\mathrm{RWH}}(x, n).$$

The inequality here depends on the fact that $|x'|$ and $|K(x)|$ are logarithmically small compared to $|x|$. This ensures that the entries in the matrices $J(K(x))$ have sizes logarithmic relative to $|x|$ and the entries in $J(xx') = J(x)J(x')$ have sizes $|J(x)| + O(\log|x|)$. These logarithms in the sizes contribute polynomials in $|x|$ as factors in the probabilities, and such factors are ignored. □

Remark 3. In [Gu2], Theorem 7.5 has been stated in a stronger form; instead of RPMC(σ) it referred to RPMC(σ, c) where the (A, S, n) were required to have $|S| = O(\log|A|)$. Our proof of the theorem does not establish the stronger result automatically. Although $|K(x)|$ is logarithmically small compared to $|x|$, we cannot conclude that $|J(K(x))|$ is logarithmically small compared to $|J(x)|$ or $|J(xx')|$, since J may shrink x or xx' much more than it shrinks $K(x)$. Since J shrinks only very few strings, the stronger form of the theorem may well be true, but we have not checked this since it does not seem to be worth the additional effort.

8. Matrix correspondence problem. In this section, a matrix is a unimodular matrix, a column is a column of two relatively prime (not necessarily positive) integers, and a matrix is seen as the pair of its columns. Call a matrix or a column positive (respectively, negative) if all its entries are nonnegative (respectively, nonpositive). If u is a column then u_1, u_2 are the upper and the lower entries of u, and $|u|$ is the positive column v such that $v_i = |u_i|$. Positive columns are ordered componentwise, as in §1. If u is a column then $\max(u) = \max(|u_1|, |u_2|)$. Any component of a column u with the absolute value $\max(u)$ is *major*, and the other component is *minor*. If X is a matrix (u, v) then $\max(X) = \max(\max(u), \max(v))$. Any entry of a matrix X with the absolute value $\max(X)$ is the *major* entry. If u, v are the two columns of a matrix X and $|u| > |v|$ then u is the *major* column and v is the *minor*; in the case of the unit matrix, both columns are *major* and both are *minor*.

LEMMA 8.1. *For every matrix* $X = (u, v)$,

1. *it is impossible that one of the numbers* u_1v_2, u_2v_1 *is positive and the other is negative. If they are both positive then* $|u_1v_2| - |u_2v_1| = 1$, *and if they are both negative then* $|u_2v_1| - |u_1v_2| = 1$;
2. *if* X *is not one of the following four matrices*

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \begin{pmatrix} -1 & 0 \\ 1 & -1 \end{pmatrix}, \quad \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \quad \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

then either $(|u|) > (|v|)$ *or* $(|u|) < (|v|)$.

Proof. (1) If one of the numbers u_1v_2, u_2v_1 is positive and the other is negative then $|u_1v_2 - u_2v_1| \geq 1 + 1$ which is impossible. If the two numbers are positive then $|u_1v_2| - |u_2v_1| = u_1v_2 - u_2v_1 = 1$, and if the two numbers are negative then $|u_2v_1| - |u_1v_2| = u_1v_2 - u_2v_1 = 1$.

(2) Suppose that X is not one of the four matrices, and suppose that neither $|u| > |v|$ nor $|v| > |u|$. Without loss of generality, $|u_1| > |v_1|$ and $|u_2| < |v_2|$; otherwise replace (u, v)

with the matrix $(v, -u)$. Since X is neither the unit matrix nor its negative, either u_2 or v_1 is not zero. Hence $|u_1v_2| - |u_2v_1| \geq (|v_1| + 1)(|u_2| + 1) - |u_2v_1| = |u_2| + |v_1| + 1 \geq 2$ which contradicts (1). □

LEMMA 8.2. *If* $\max(u, v) > 1$ *then* (u, v) *has only one major entry.*

Proof. The proof is very similar to the proof of the second part of Lemma 6.1. There are some minor differences however, and—for the reader's convenience—we present the proof. By contradiction suppose that $m = \max(u, v) > 1$ but (u, v) has two or more major entries. If two major entries occur in the same row or column then m divides the determinant, which is impossible. Thus, there are exactly two major entries. Since (u, v) may be replaced with $(v, -u)$, we may suppose that the two major entries form the second diagonal, i.e., $|u_2| = |v_1| = m$. If $u_2v_1 > 0$ then the determinant is negative which is impossible. Hence $u_2v_1 < 0$. By Lemma 8.1, $u_1v_2 \leq 0$ and $1 = |u_2v_1| - |u_1v_2| \geq m^2 - (m-1)^2 = 2m - 1 \geq 3$, which is false. □

LEMMA 8.3. *For every two matrices* (u, v) *and* (u, v')*, there exists an integer* k*, such that* $v' = v + ku$.

Proof. $u_1v_2' - u_2v_1' = 1 = u_1v_2 - u_2v_1$ and therefore $u_1(v_2' - v_2) = u_2(v_1' - v_1) = u_1u_2k$ for some k. If neither component of u is zero, the claim is obvious. Suppose that one of the components of u is zero. By symmetry, let $u_1 = 0$. Then $v_1' = v_1$, $|u_2| = 1$ and the claim is clear. □

LEMMA 8.4. *Let* $X = (u, v)$ *be any matrix with* $\max(X) > 1$. *If* u *(respectively,* v*) is the major column of* X *then there exists exactly one additional matrix of the form* (u, v') *(respectively,* (u', v)*) where the column* v' *(respectively,* u'*) is minor. Moreover,* $v' = v \pm u$ *(respectively,* $u' = u \pm v$*). If the major column is positive or negative then one of the two possible minor columns is positive and the other one is negative.*

Proof. It suffices to consider the case when u is the major column because if (u, v) is a counterexample with the major column on the right then $(-v, u)$ is a counterexample with the major column on the left. Further, it suffices to consider the case when the major entry is positive because if (u, v) is a counterexample with a negative major entry then $(-u, -v)$ is a counterexample with a positive major entry. Let u_i be the major entry of u and (u, v') be another matrix with major column u. By Lemma 8.3, $v' = v + ku$ for some k. Since $u_i > 1$, $v_i \neq 0$. If $v_i > 0$ then $k = -1$, and if $v_i < 0$ then $k = 1$. Note also that if $v_i > 0$ (respectively, $v_i < 0$) then indeed u is the major column of the matrix $(u, v - u)$ (respectively, $(u, v + u)$). Now suppose that u is positive. Obviously, $u_1 > 0$ and $u_2 > 0$. By part 1 of Lemma 6.1, v is either negative or positive. If v is positive (respectively, negative) then v' is negative (respectively, positive). □

Let $SL_2(Z)$ denote not only the modular group but also the uniform domain of unimodular matrices with $|X| = \ell(\max(X))$.

LEMMA 8.5. *Let* $m > 1$ *and* X *be a random unimodular matrix with* $\max(X) = m$. *The probability that* X *is positive is* 1/8*, and the probability that* X *is the inverse of a positive matrix is* 1/8 *as well.*

Proof. Let S_0 be the collection of matrices X with $\max(X) = m$. The inverse of a matrix $\begin{pmatrix} a & c \\ b & d \end{pmatrix}$ is the matrix $\begin{pmatrix} d & -c \\ -b & a \end{pmatrix}$; thus $\max(X^{-1}) = \max(X)$ and therefore S_0 is closed under inversion. It follows that the number of positive matrices in S_0 equals the number of the inverses of positive matrices. Hence it suffices to prove only the first statement of the lemma.

Let S_1 be the collection of S_0 matrices X such that the major entry of X is positive. For every (u, v) in S_0, exactly one of the two matrices (u, v), $(-u, -v)$ belongs to S_1. It remains to prove that the probability of a random S_1 matrix being positive is 1/4.

Since the major entry of an S_1 matrix exceeds 1, the minor component of the major column is not zero. Let S_2 be the collection of S_1 matrices such that the minor component of the major

column is positive. For every S_1 matrix X, let X' be the result of multiplying by -1 the diagonal of X that contains the minor component of the major column. Exactly one of the two matrices X, X' belongs to S_2. It follows that S_2 contains exactly one half of the elements of S_1. It remains to prove that the probability of a random S_2 matrix being positive is 1/2. Now use Lemma 8.4. □

The direct product $\mathrm{SL}_2(Z) \times \mathrm{SL}_2(Z)$ is a domain and monoid of matrix pairs; the multiplication of matrix pairs is componentwise: $(X_1, Y_1) \times (X_2, Y_2) = (X_1X_2, Y_1Y_2)$. If S is a set of matrix pairs, let S^n comprise products $P_1 \times \cdots \times P_m$ where $m \leq n$ and each $P_i \in S$. Let σ be a positive integer. In the following definition, MC stands for Matrix Correspondence.

DEFINITION 8.6. *For each positive integer σ, $\mathrm{MC}(\sigma)$ is the decision problem with domain $\mathrm{SL}_2(Z) \times \mathrm{Set}_\sigma(\mathrm{SL}_2(Z) \times \mathrm{SL}_2(Z)) \times \mathrm{PI}$ where an instance (A, S, n) is positive if and only if there exists a pair $(X, Y) \in S^n$ such that $AX = Y$.*

Let σ witness Theorem 7.5.

THEOREM 8.7. *$\mathrm{MC}(\sigma)$ is hard for RNP. Moreover, so is its restriction to the subdomain of those instances (A, S, n) where each pair in S consists of positive matrices.*

Proof. The identity function reduces $\mathrm{PMC}(\sigma)$ to the desired restriction of $\mathrm{MC}(\sigma)$ and therefore to $\mathrm{MC}(\sigma)$ itself. To check the domination property, use Lemmas 3.7 and 8.5. □

9. Linear transformations of the modular group.

THEOREM 9.1. *Suppose that $T : \mathrm{SL}_2(Z) \to \mathrm{SL}_2(Z))$ is linear in the sense that, if $X = \sum_{i=1}^{k} Y_i$ with X and all Y_i in $\mathrm{SL}_2(Z)$, then $T(X) = \sum_{i=1}^{k} T(Y_i)$. Then there exist B and C in $\mathrm{SL}_2(Z)$ such that either, for all $X \in \mathrm{SL}_2(Z)$, $T(X) = BXC$ or, for all such X, $T(X) = BX^tC$ where the superscript t denotes transpose.*

Proof. We first normalize T so that $T(I) = I$, where I is the identity matrix. If the given T does not fix I, then we consider T' given by $T'(X) = T(I)^{-1}T(X)$, and we observe that the hypotheses of the theorem about T imply the same hypotheses about T' and the conclusion about T' (with $C = B^{-1}$) implies the same conclusion about T. Thus we may as well work with T', which fixes I, instead of T. So, from now on, we assume that $T(I) = I$.

Notation. $\mathrm{M}_2(Q)$ (respectively, $\mathrm{M}_2(C)$) is the vector space of two-by-two matrices with rational (respectively, complex) entries. As usual, $\mathrm{SL}_2(Q)$ (respectively, $\mathrm{SL}_2(C)$) is the multiplicative group of two-by-two matrices X with rational (respectively, complex) entries such that $\det(X) = 1$.

Our next goal is to show that the linearity hypothesis on T implies that T can be extended to a linear transformation (in the usual sense) on $\mathrm{M}_2(Q)$. Let $\mathcal{B}$ be the set of the following four matrices in $\mathrm{SL}_2(Z)$:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I, \quad \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 1 \\ -1 & 0 \end{pmatrix}.$$

It is easy to see that every matrix in the standard basis for $\mathrm{M}_2(Q)$

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \quad \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \quad \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}, \quad \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

is a linear combination (with integer coefficients) of the $\mathcal{B}$ matrices, so that $\mathcal{B}$ is a basis for $\mathrm{M}_2(Q)$. There is no question what the linear extension of T should be; it is the unique linear transformation $\bar{T}$ that agrees with T on these four basis matrices. Our task is to show that it agrees with T on the other matrices in $\mathrm{SL}_2(Z)$ as well. Any $\mathrm{SL}_2(Z)$ matrix X is a linear combination, with rational coefficients, of the four $\mathrm{SL}_2(Z)$ matrices in the basis; to show that $T(X) = \bar{T}(X)$, it suffices to prove that $T(X)$ is the similar linear combination of T-images of the four matrices. Thus, it suffices to show that any linear dependence relation with rational

coefficients that holds between some matrices in $SL_2(Z)$ also holds between their T-images. Furthermore, we may suppose that the coefficients are integers (since we can multiply by a common denominator of the rational coefficients) and in fact that the coefficients are all ± 1 (as other coefficients can be replaced by repeated terms). Comparing what we need to prove with the hypothesis of the theorem, we find that we need only check that $T(-X) = -T(X)$ for all $X \in SL_2(Z)$. But this is easy; just apply the hypothesis to the linear relation $X = -X + X + X$.

From now on, we write T not only for the given function but also for the corresponding linear transformation of $M_2(Q)$ (called $\bar{T}$ above), and for the unique extension of this to a linear transformation of $M_2(C)$.

We note for future reference that if a matrix X from $M_2(C)$ has integer entries then so does $T(X)$. Indeed, this claim is true by hypothesis if X has determinant 1 and in particular for the four $\mathcal{B}$ matrices. By linearity of T, the claim follows for any X that is a linear combination with integer coefficients of $\mathcal{B}$ matrices. In particular the claim is true for the four matrices in the standard basis of $M_2(Q)$ and therefore it is true for all X in $M_2(C)$.

We also need that T preserves determinants, i.e., that $\det T(X) = \det(X)$ for all $X \in M_2(C)$. This is true by hypothesis if $X \in SL_2(Z)$, but some work will be needed to extend it to more general X.

Begin by considering $X \in M_2(Q)$. For such an X, the following two conditions are equivalent: (1) The determinant and the trace of X both vanish; (2) There are at least two distinct nonzero rational numbers r for which $I + rX \in SL_2(Z)$. This follows from the formula

$$\det(I + rX) = 1 + r \cdot \operatorname{tr}(X) + r^2 \cdot \det(X).$$

If (1) holds, then $I + rX$ has determinant 1 for all r, so we can satisfy (2) by taking any two r's for which the entries of rX are integers. Conversely, if (2) holds then we have two linear equations satisfied by the determinant and the trace of X, namely

$$r \cdot \operatorname{tr}(X) + r^2 \cdot \det(X) = 0$$

for each of the two r's. As the two equations are linearly independent, (1) follows.

It is clear, from inspection of condition (2), that if X satisfies it then so does $T(X)$. Thus, T maps the set N of matrices satisfying (2) or, equivalently, (1) into itself. T therefore also maps the linear span $\bar{N}$ of N into itself. Note that the trace of every $\bar{N}$ matrix is zero and that matrices with zero trace form a three-dimensional subspace of $M_2(Q)$. Since N contains the matrices

$$\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \quad \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}, \quad \text{and} \quad \begin{pmatrix} 1 & 1 \\ -1 & -1 \end{pmatrix},$$

it follows that $\bar{N}$ is exactly that three-dimensional subspace. In this three-dimensional vector space $\bar{N}$, N is a cone, the zero-set of the quadratic form det.

We check that, for some k, $\det(T(X)) = k \cdot \det(X)$ for all X of trace zero. Let X be

$$\begin{pmatrix} x & y \\ z & -x \end{pmatrix},$$

so that $\det(X) = -x^2 - yz$. As T is linear and det is quadratic,

$$\det(TX) = \alpha x^2 + \beta y^2 + \gamma z^2 + \delta xy + \varepsilon xz + \zeta yz$$

for some coefficients $\alpha, \beta, \gamma, \delta, \varepsilon, \zeta$. Since T maps N onto itself, $\det(TX) = 0$ if $\det(X) = 0$. Choose $y = x^2$ and $z = -1$ so that $\det(X) = 0$ and therefore $\det(TX) = 0$. We have that,

for all x, $\alpha x^2 + \beta x^4 + \gamma + \delta x^3 - \varepsilon x - \zeta x^2 = 0$, so that $\alpha = \zeta$ and $\beta = \gamma = \delta = \varepsilon = 0$. Thus, $\det(TX) = \alpha x^2 + \alpha yz = -\alpha \det(X)$.

Consider in particular the matrices

$$X = \begin{pmatrix} 0 & 3 \\ 1 & 0 \end{pmatrix} \quad \text{and} \quad 2I + X = \begin{pmatrix} 2 & 3 \\ 1 & 2 \end{pmatrix},$$

and note that X has trace zero while $2I + X \in \mathrm{SL}_2(Z)$. So we have

$$\begin{aligned} 1 &= \det(T(2I + X)) \\ &= \det(2I + T(X)) \\ &= 4 + 2 \cdot \mathrm{tr}(T(X)) + \det(T(X)) \\ &= 4 + 2 \cdot 0 + k \cdot \det(X) \\ &= 4 - 3k, \end{aligned}$$

so $k = 1$ and $\det(T(X)) = \det(X)$ for all rational X of trace zero.

For rational X of arbitrary trace, we can write $X = rI + Y$ where r is rational and $Y \in \bar{N}$. Then by what we have already proved, $T(Y)$ has trace zero and the same determinant as Y. So

$$\begin{aligned} \det(T(X)) &= \det(T(rI + Y)) \\ &= r^2 + r \cdot \mathrm{tr}(T(Y)) + \det(T(Y)) \\ &= r^2 + \det(Y) \\ &= \det(rI + Y) = \det(X). \end{aligned}$$

This shows that T preserves determinants of rational matrices. It follows that it preserves determinants of real matrices (by continuity) and of complex matrices (by analytic continuation). (Here is a more elementary argument. We have the equation $\det(T(X)) = \det(X)$ when all four entries are rational. If we let one entry, say the upper left one, vary over complex numbers while the other three entries remain fixed rational numbers, then the equation remains true because a polynomial equation in one variable that holds infinitely often must hold identically. Then we let another entry vary, while the remaining three stay fixed, one being an arbitrary complex number and the other two rational. Repeating the process for each entry in turn, we find that the equation holds for all complex values of the entries.)

Summarizing what we have achieved so far, we have a linear, determinant-preserving transformation T on $\mathrm{M}_2(C)$, which sends I to itself and sends integer matrices to integer matrices. Our immediate goal is to show that there is a matrix $B \in \mathrm{SL}_2(C)$ satisfying the conclusion of the theorem with $C = B^{-1}$ (as $T(I) = I$), not only for all $X \in \mathrm{SL}_2(Z)$ but for all $X \in \mathrm{M}_2(C)$. (This information is essentially contained in [W, pp. 19–21], but for the reader's convenience we give a different, more detailed proof.) Once this is done, we will complete the proof by showing that the entries of B must be integers.

Until we reach our intermediate goal, we shall be working in the complex vector space $\mathrm{M}_2(C)$, and it will be convenient to use the following basis for this space:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad P = \begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix}, \quad Q = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \quad R = \begin{pmatrix} 0 & i \\ i & 0 \end{pmatrix}.$$

The advantage of this basis is that the determinant is given by a very simple formula

$$\det(aI + pP + qQ + rR) = \det \begin{pmatrix} a + pi & q + ri \\ -q + ri & a - pi \end{pmatrix} = a^2 + p^2 + q^2 + r^2.$$

As T is linear and preserves determinants and I, it preserves eigenvalues; indeed, if $X - xI$ has determinant zero then so does $T(X - xI) = T(X) - xI$. In particular, $T\left(\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right)$ has eigenvalues 1 and 0; viewed as a transformation of 2-component vectors, it is a projection onto some line along some other line, which means that it has the form

$$T\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} p \\ q \end{pmatrix}(r \quad s) = \begin{pmatrix} pr & ps \\ qr & qs \end{pmatrix}$$

for some p, q, r, s. Furthermore, since the eigenvalues are 1 and 0, the trace is 1, so $pr + qs = 1$. This means that the matrix

$$A = \begin{pmatrix} p & -s \\ q & r \end{pmatrix}$$

has determinant 1. The transformation $X \mapsto AXA^{-1}$ sends I to itself and sends $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right)$ to $\left(\begin{smallmatrix} pr & ps \\ qr & qs \end{smallmatrix}\right)$, just like T. So the linear transformation

$$T'(X) = A^{-1}T(X)A$$

preserves determinants and fixes both I and $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right)$ and therefore also their linear combination P. If we achieve our intermediate goal for T', the same result will follow immediately for T. Indeed, if $T'(X) = BXB^{-1}$ then $T(X) = (AB)X(AB)^{-1}$, while if $T'(X) = BX^tB^{-1}$ then $T(X) = (AB)X^t(AB)^{-1}$. So we may work with T' instead of T.

Thus, we assume that T fixes both I and P. Furthermore, as T preserves the quadratic form det, it also preserves the associated bilinear form

$$\langle X, Y\rangle = \tfrac{1}{2}(\det(X + Y) - \det(X) - \det(Y)),$$

which has, relative to our chosen basis, the standard form

$$\langle aI + pP + qQ + rR, a'I + p'P + q'Q + r'R\rangle = aa' + pp' + qq' + rr'.$$

(Usually, complex linear spaces are equipped with inner products that are linear in one factor and conjugate-linear in the other. That is not the case here; our inner product is linear in both factors.)

As T preserves this inner product and fixes I and P, it must leave invariant the set of vectors orthogonal to both I and P, namely the linear span of Q and R. So we have $T(Q) = qQ + rR$ for some scalars q and r. Also, we have

$$1 = \det(Q) = \det(T(Q)) = \det(qQ + rR) = q^2 + r^2.$$

There is a complex number v such that $(v + (1/v))/2 = q$. (Just solve a quadratic equation for v; of course there is a second solution $1/v$.) Note that

$$\left[\frac{1}{2} = \left(v + \frac{1}{v}\right)\right]^2 + \left[\frac{1}{2i}\left(v - \frac{1}{v}\right)\right]^2 = 1 = q^2 + r^2,$$

so $r = \pm(v - (1/v))/2i$. Replacing v with $1/v$ if necessary, we can arrange that

$$q = \frac{1}{2}\left(v + \frac{1}{v}\right) \quad \text{and} \quad r = \frac{1}{2i}\left(v - \frac{1}{v}\right).$$

Let u be either of the square roots of v, and let

$$M = \begin{pmatrix} u & 0 \\ 0 & \frac{1}{u} \end{pmatrix}.$$

Note that

$$M \cdot \begin{pmatrix} x & y \\ z & w \end{pmatrix} \cdot M^{-1} = \begin{pmatrix} x & u^2 y \\ \frac{1}{u^2} z & w \end{pmatrix} = \begin{pmatrix} x & vy \\ \frac{1}{v} z & w \end{pmatrix}.$$

In particular, the transformation $X \mapsto MXM^{-1}$ fixes I and P (just as T does) and it sends $Q = \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$ to

$$\begin{aligned} \begin{pmatrix} 0 & v \\ -\frac{1}{v} & 0 \end{pmatrix} &= \frac{1}{2}\left(v + \frac{1}{v}\right)\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} + \frac{1}{2i}\left(v - \frac{1}{v}\right)\begin{pmatrix} 0 & i \\ i & 0 \end{pmatrix} \\ &= qQ + rR \\ &= T(Q). \end{aligned}$$

So $T'(X) = M^{-1}T(X)M$ preserves determinants and fixes I, P, and Q. As before, it suffices to reach our intermediate goal for T' rather than T. So from now on we assume that T fixes I, P, and Q.

It follows that T fixes the subspace orthogonal to I, P, and Q, namely the subspace spanned by R. So $T(R)$ is a scalar times R, and the scalar can only be ± 1 because T preserves determinants. If the scalar is 1, then T fixes all four basis matrices, hence is the identity. If, on the other hand, the scalar is -1, then $T(X) = P^{-1}X^tP$, because the right side of this equation defines a linear transformation which, like T fixes I, P, and Q and reverses the sign of R. In either case, T has the required form, so we have achieved our intermediate goal.

We now return to the original T, normalized to fix I and extended to $\mathrm{M}_2(C)$, which we now know to have the form $T(X) = BXB^{-1}$ or $T(X) = BX^tB^{-1}$ for some $B \in \mathrm{SL}_2(C)$. We also know that if the entries of X are integers then so are those of $T(X)$. What we still need to show is that the entries of B are integers. Without loss of generality, we may suppose that $T(X) = BXB^{-1}$.

Let X be the matrix with a single entry equal to 1, say the entry in position i, j, and all other entries zero. Then the entries of $T(X)$, namely

$$(BXB^{-1})_{k,l} = B_{k,i}(B^{-1})_{j,l}$$

are integers for all k and l. But, as B has determinant 1, the entries of B^{-1} are the same as those of B, except for their signs and their positions in the matrices. Thus we see that the product of any two entries of B is an integer.

In particular, the square of each entry of B is an integer, so each entry is the product of an integer and (possibly) the square roots of certain distinct primes.

Suppose p is a prime whose square root occurs in one of the entries. Then $\sqrt{p}$ must occur in every entry, for the product of an entry containing $\sqrt{p}$ as a factor and another entry not containing it could never be an integer. So $\sqrt{p}$ occurs as a factor of every entry of B. But then p is a factor of $\det(B) = 1$. This contradiction shows that no square roots occur.

So every entry of B is an integer, and the proof is complete. □

We saw that an arbitrary linear transformation T over $\mathrm{SL}_2(Z)$ extends uniquely to a linear transformation over the vector space $M_2(R)$ of all four-by-four real matrices. Let $\mathrm{Mat}(T)$ be the matrix of (the extension of) T in the standard basis:

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \quad \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \quad \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}, \quad \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

LEMMA 9.2. *We have*

$$T\begin{pmatrix} x_1 & x_2 \\ x_3 & x_4 \end{pmatrix} = \begin{pmatrix} y_1 & y_2 \\ y_3 & y_4 \end{pmatrix} \longleftrightarrow \mathrm{Mat}(T)\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}.$$

Proof. The proof is obvious. □

The entries of $\mathrm{Mat}(T)$ are the entries of the four matrices obtained by applying T to the standard basis. By Theorem 9.1, these are integers.

By Theorem 9.1, there are unimodular matrices B and C such $T(X) = BXC$ for all unimodular matrices X or $T(X) = BX^tC$ for all unimodular matrices X. Call T *right* in the first case and *left* in the second. The determinant of $\mathrm{Mat}(T)$ is ± 1 because

$$\det(\mathrm{Mat}(T)) \times \det(\mathrm{Mat}(T^{-1})) = \det(T \times T^{-1}) = 1.$$

LEMMA 9.3. *Right transformations are exactly those with determinant* $+1$.

Proof. It suffices to prove that if T is right then $\det \mathrm{Mat}(T) = 1$, because a left T is the product of the transposing transformation (whose matrix has determinant 1) and a right transformation. So suppose that $T(X) = BXC$ for some unimodular matrices B, C and all unimodular (and therefore all two-by-two real matrices) X.

Now forget about unimodular matrices and think about real matrices. Every pair (B', C') of nonsingular two-by-two real matrices gives a linear transformation $T' = X \mapsto B'XC'$ over two-by-two real matrices that has an inverse, so that the matrix $\mathrm{Mat}(T')$ of T' in the standard bases is nonzero. Now continuously transform B and C to the unit matrix. In the process T is continuously transformed to the identity, whose matrix has determinant 1, and the determinant of T remains nonzero all the time. Therefore its initial value cannot be -1. □

LEMMA 9.4. *There is a PTime algorithm that, given a four-by-four integer matrix* M, *determines whether* $M = \mathrm{Mat}(T)$ *for some* T.

Proof. If $M = \mathrm{Mat}(T)$ then, by Theorem 9.1, there exist unimodular matrices B and C such that either $T(X) = BXC$ for all unimodular matrices X or $T(X) = BX^tC$ for all unimodular matrices X. We show how to find out whether there is a pair (B, C) satisfying the first condition and even how to find all pairs (B, C) satisfying the first condition. The case of the second condition is similar.

Suppose that $T(X) = BXC$ for all unimodular matrices X. Due to the unique extendibility of T, $T(X) = BXC$ for all two-by-two real matrices. Let

$$B = \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix}, \quad C = \begin{pmatrix} c_1 & c_2 \\ c_3 & c_4 \end{pmatrix}.$$

Computing the products BXC where X belongs to the standard basis and using Lemma 9.2, we have

$$M = \begin{pmatrix} b_1c_1 & b_1c_3 & b_2c_1 & b_2c_3 \\ b_1c_2 & b_1c_4 & b_2c_2 & b_2c_4 \\ b_3c_1 & b_3c_3 & b_4c_1 & b_4c_3 \\ b_3c_2 & b_3c_4 & b_4c_2 & b_4c_4 \end{pmatrix}.$$

Thus we can recover all b_i/b_j and all c_i/c_j. So we recover B and C up to scalar factor. We recover the scalar factor, except for the sign, using the equalities $\det(B) = \det(C) = 1$. It follows that the pair (B, C) is unique except for an over-all sign. □

10. Matrix transformation. In this section we prove that Matrix Transformation is hard for RNP.

For an arbitrary numerical matrix X, let $\max(X)$ be the maximal absolute value of the entries of X. Recall that $\ell(n)$ is the length of the binary notation for n and $\mathrm{Mat}(T)$ is the matrix of a linear transformation T.

DEFINITION 10.1. *LT (standing for "Linear Transformations") is the uniform domain of linear transformations of* $\mathrm{SL}_2(Z)$ *with* $|T| = \ell(\max(\mathrm{Mat}(T)))$.

It will be convenient to ignore the distinction between a linear transformation T over $\mathrm{SL}_2(Z)$ and its matrix $\mathrm{Mat}(T)$.

For all unimodular matrices B and C, let $T_{B,C}$ be the linear transformation $X \mapsto C^{-1}XB$.

LEMMA 10.2. *The function* $(B, C) \mapsto M_{B,C}$ *reduces the subdomain of positive pairs in* $\mathrm{SL}_2(Z) \times \mathrm{SL}_2(Z)$ *to LT.*

Proof. We need to check only that the function $f(B, C) = M_{B,C}$ has the domination property. Recall that the inverse of a matrix $\left(\begin{smallmatrix} a & c \\ b & d \end{smallmatrix}\right)$ is the matrix $\left(\begin{smallmatrix} d & -c \\ -b & a \end{smallmatrix}\right)$. It follows that $\max(f(B, C)) = \max(B) \times \max(C)$ and therefore $|B| + |C| - 1 \le |f(B, C)| \le |B| + |C|$.

Let $l = |f(B, C)|$, let M range over LT, and let B, C range over unimodular matrices. Using # as the cardinality symbol, we have

$$\begin{aligned} &\#\{M : |M| = l\} \le 2\#\{(B, C) : l \le |B| + |C| \le l + 1\} \\ &= \sum_{j=1}^{l+1} [\#\{B : |B| = j\} \times \#\{C : l - j \le |C| \le l - j + 1\}]. \end{aligned}$$

By Lemma 8.5, the number of unimodular matrices of size $m > 1$ is eight times the number of positive unimodular matrices of size m. According to Lemma 6.7, the later number is $\Theta(2^{2m})$. It follows that, modulo a constant factor,

$$\#\{M : |M| = l\} \le \sum_{l=1}^{l+1} 2^{2j} 2^{2l-2j} = 2^{2l}(l + 1).$$

We saw in the previous section that, for each M in LT, the pre-image of $f^{-1}(M)$ has at most two elements. It follows that, modulo bounded factors,

$$\frac{\mathbf{P}(f^{-1}f(B, C))}{\mathbf{P}(f(B, C))} \le \frac{2^{2l}(l + 1)}{2^{2l}} = l + 1 \le |(B, C)| + 1$$

which is AP on $\mathrm{SL}_2(Z) \times \mathrm{SL}_2(Z)$. □

If S is a subset of LT, let S^n be the set of products $T_m \cdots T_1$ where $m \le n$ and each $T_i \in S$. First we prove that an auxiliary version of matrix transformation is hard for RNP.

DEFINITION 10.3. *For each positive integer* σ, $\mathrm{MT}(\sigma)$ *is the decision problem with domain* $\mathrm{SL}_2(Z) \times \mathrm{Set}_\sigma(LT) \times PI$ *where an instance* (A, S, n) *is positive if and only if there exists* $T \in S^n$ *that transforms* A *to the unit matrix.*

THEOREM 10.4. *Some* $\mathrm{MC}(\sigma)$ *is hard for RNP.*

Proof. Let σ witness Theorem 8.7. We reduce the subdomain of $\mathrm{MC}(\sigma)$ described in Theorem 8.7 to $\mathrm{MC}(\sigma)$. If S is a sequence of positive matrix pairs, let S' be the result of replacing each pair (B, C) in S with the linear transformation $T_{B,C}(X) = C^{-1}XB$. The desired reduction is $f(A, S, n) = (A, S', n)$. To check the correctness property, note that $AB_1 \ldots B_m = C_1 \ldots C_m$ if and only if the transformation $T_{B_m,C_m} \ldots T_{B_1,C_1}$ takes A to the unit matrix.

It remains to check that f reduces the relevant subdomain of $\mathrm{SL}_2(Z) \times (\mathrm{SL}_2(Z) \times \mathrm{SL}_2(Z))^\sigma \times \mathrm{PI}$ to the domain $\mathrm{SL}_2(Z) \times (\mathrm{LT})^\sigma \times \mathrm{PI}$. It suffices to check that the function

$S \mapsto S'$ reduces $(\mathrm{SL}_2(Z) \times \mathrm{SL}_2(Z))^\sigma$ to LT^σ. By Lemma 3.7, it suffices to check that the function $(B, C) \mapsto T_{B,C}$ reduces $\mathrm{SL}_2(Z) \times \mathrm{SL}_2(Z)$ to LT. Now use Lemma 10.2. □

DEFINITION 10.5. *MT is the decision problem with domain* $\mathrm{SL}_2(Z) \times \mathrm{Set}(LT) \times PI$ *such that an instance* (A, S, n) *is positive if and only if there exists* $P \in S^n$ *with* $A = P(1)$.

COROLLARY 10.6. *MT is RNP complete.*

Proof. The identity function deterministically reduces $\mathrm{MT}(\sigma)$ to MT. We omit checking that MT is in RNP. □

Remark 4. Let $\pi(j)$ be any PTime computable nondecreasing function from positive integers to positive integers such that the inverse function $\pi^{-1}(j) = \min_i[\pi(i) \geq j]$ is polynomially bounded. For example, $\pi(j) = j$. The restriction of MT (respectively, $\mathrm{MT}(\sigma)$) to instances (A, S, n) with $n = \pi(|A|)$ remains RNP complete. The proof is the same proof except we start with the corresponding version of the bounded halting problem, which has been proved RNP complete in [Gu1, §9].

11. Bounded membership problem. In this section, we briefly consider a natural simplification of $\mathrm{MT}(\sigma)$ where the question is whether the given unimodular matrix X is a product of at most n factors taken from a given finite subset (assumed closed under inverses) of $\mathrm{SL}_2(Z)$. This is a bounded version of the membership problem [Mi] for $\mathrm{SL}_2(Z)$. We show that it is NP complete. It is interesting open problem whether a natural randomization of it is RNP complete. We begin with the analogous bounded version of the membership problem for the additive group of integers.

DEFINITION 11.1. Integer Sum *is the following* NP *problem:*

Instance: *A positive integer* K, *a finite set* S *of positive integers, and a positive integer* n.

Question: *Can* K *be represented as* $\sum_{i=1}^{m} \varepsilon_i b_i$ *where* $m \leq n$, *the numbers* b_i *are (not necessarily distinct) elements of* S, *and* $\varepsilon_i \in \{1, -1\}$?

The restriction n on the number of summands is important. It is easy to decide whether or not K can be represented as a sum of elements of $S \cup \{-b : b \in S\}$; just compute the greatest common divisor of the elements of S and check whether it divides K.

LEMMA 11.2. *Integer Sum is* NP *complete.*

The fact may be well known. It was not known to us. Suzanne Zeitman, a graduate student of Gurevich, proved the lemma.

Proof. The proof is by reduction from X3C, Exact Cover by 3-Sets [GJ], which is the following NP problem:

Instance: A positive integer q and a collection C of 3-element subsets of the set $\{1, 2, \ldots, 3q\}$.

Question: Is there an exact cover $C' \subseteq C$ for X (so that each element of X belongs to exactly one member of C')?

The transformation f we use resembles that used in the reduction of 3-Dimensional Matching to partition [GJ]. Given an instance (q, C) of X3C, let l be the length of the binary notation for q and B be the collection of binary strings of length $3ql$. View a B string as a sequence of $3q$ blocks (substrings) of length l. For each i, $1 \leq i \leq 3q$, let a_i be the integer represented by a B string with exactly one 1, which appears at the rightmost position of the ith block.

CLAIM 11.3. *If* $\sum_{i=1}^{3q} \alpha_i a_i = \sum_{i=1}^{3q} \beta_i a_i$ *and* $0 \leq \alpha_i, \beta_i \leq q$ *for each* i, *then* $\alpha_i = \beta_i$ *for each* i.

Proof. By the definition of l. □

Define

$$f(q, C) = (K, \{y(T) : T \in C\}, q),$$

where $K = \sum_{i=1}^{3q} a_i$ and each $y(T) = \sum_{i \in T} a_i$. If C' is an exact cover of C, then the cardinality of C' is q and K is represented as the sum of the q numbers $y(T)$ such that $T \in C'$.

Now suppose that $K = \sum_{j=1}^{m} \varepsilon_j y(T_j)$ where $m \leq q$ and $\varepsilon_j \in \{1, -1\}$ and $T_j \in C$. Then let $K^+ = \sum\{y(T_j) : \varepsilon_j > 0\} = \sum_{i=1}^{3q} \alpha_i a_i$ and $K^- = \sum\{y(T_j) : \varepsilon_j < 0\} = \sum_{i=1}^{3q} \gamma_i a_i$, so that $K^+ = K + K^-$. Clearly, $\alpha_i \leq m \leq q$. Similarly, $\gamma_i \leq q$. For each $T \in C$, let $z_i(T)$ equal 1 if $i \in T$ and equal 0 otherwise.

First consider the case $K^- = 0$. By Claim 11.3, we have that, for each i, $\sum_j z_i(T_j) = 1$. Thus, the sets T_j form an exact cover.

By contradiction, suppose that $K^- > 0$. Then the number of j's with $\varepsilon_j > 0$ is less than q and therefore there exists an i with $\alpha_i = 0$. Since $K^+ = K + K^-$, we have, by the claim, that $0 = \alpha_i = 1 + \gamma_i$ which is impossible. □

DEFINITION 11.4. *The* bounded membership problem *for the modular group, in short BM, is the following* NP *decision problem:*

Instance: *A unimodular matrix X, a finite set S of unimodular matrices and a positive integer n.*

Question: *Can X be represented as $\prod_{i=1}^{m} Y_i$ where $m \leq n$ and, for each i, either Y or Y^{-1} is in S?*

COROLLARY 11.5. *BM is* NP *complete.*

Proof. For each integer y, let

$$g(y) = \begin{pmatrix} 1 & y \\ 0 & 1 \end{pmatrix}.$$

If $g(y) = Y$ and $g(z) = Z$ then $g(y + z) = YZ$ and $g(-y) = Y^{-1}$. This gives rise to the following reduction of IS to BM:

$$F(K, S, n) = (g(K), \{g(y) : y \in S\}, n). \qquad \square$$

One natural way to randomize BM is to view the domain of BM as $\mathrm{SL}_2(Z) \times \mathrm{Set}(\mathrm{SL}_2(Z)) \times$ PI. The corresponding randomized decision problem is probably decidable in AP time.

Acknowledgments. We thank Suzanne Zeitman for allowing us to publish her proof that Integer Sum is NP complete. We thank Abraham Sharell, the team of Belanger and Wang, and the referees for pointing out various flaws in the previous versions of this paper.

REFERENCES

[BCGL] S. BEN-DAVID, B. CHOR, O. GOLDREICH, AND M. LUBY, *On the theory of average case complexity*, 21st Annual ACM Symposium on Theory of Computing, ACM, 1989, pp. 204–216.

[BG1] A. BLASS AND Y. GUREVICH, *On the reduction theory for average-case complexity*, CSL'90, 4th Workshop on Computer Science Logic, Heidelberg, Germany, Lecture Notes in Computer Sci., Vol. 533, 1991, pp. 17–30.

[BG2] ———, *Randomizing reductions of search problems*, SIAM J. Comput., to appear. An extended abstract in FST&TCS'91, 11th Conference on Foundations of Software Technology and Theoretical Computer Science, New Delhi, India, Springer Lecture Notes in Computer Sci., Vol. 560 (1991), pp. 10–24.

[BG3] ———, *Randomized Reductions of Decision Problems* (tentative title), in preparation.

[Ei] S. EILENBERG, *Automata, Languages, and Machines*, Vol. A and B, Academic Press, NY & London, 1974 and 1976.

[GJ] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, New York, 1979.

[Gu1] Y. GUREVICH, *Average Case Completeness*, J. Comput. System Sci., 42 (1991), pp. 346–398. (An extended abstract in FOCS'87.)

[Gu2] Y. GUREVICH, *Matrix Decomposition Problem is Complete for the Average Case*, FOCS'90, 31st Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1990, pp. 802–811.

[Gu3] Y. GUREVICH, *Average case complexity*, ICALP'91, 18th International Colloquium on Automata, Languages and Programming, Madrid, Springer Lecture Notes in Computer Sci., Vol. 510, 1991, pp. 615–628.

[GS] Y. GUREVICH AND S. SHELAH, *Expected computation time for Hamiltonian path problem*, SIAM J. Comput., 16 (1987), pp. 486–502.

[HW] G. H. HARDY AND E. M. WRIGHT, *An Introduction to the Theory of Numbers*, Oxford University Press, 1988.

[IL] R. IMPAGLIAZZO AND L. A. LEVIN, *No Better Ways to Generate Hard NP Instances than Picking Uniformly at Random*, 31st Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1990.

[Jo] D. S. JOHNSON, *The NP-Completeness Column*, J. Algorithms 5 (1984), pp. 284-299.

[Kn1] D. E. KNUTH, *The Art of Computer Programming*, Vol. 1, 2nd ed., Addison-Wesley, Reading, MA, 1973.

[Kn2] ———, *Big omicron and big omega and big theta*, SIGACT News, Apr.–June, 1976, pp. 18–24.

[Le] L. A. LEVIN, *Average Case Complete Problems*, SIAM J. Comput., 15 (1986), pp. 285–286.

[Mi] C. F. MILLER, III, *Decision Problems in Algebraic Classes of Groups (a Survey)*, in Word Problems, W. W. Boone, F.B. Cannonito, and R.C. Lyndon, eds. North-Holland, Amsterdam, 1973, pp. 507–523.

[VL] R. VENKATESAN AND L. LEVIN, *Random instances of a graph coloring problem are hard*, 20th Symp. on Theory of Computing, ACM, 1988, pp. 217–222.

[VR] R. VENKATESAN AND S. RAJAGOPALAN, *Average case intractability of matrix and diophantine problems*, 24th Symp. on Theory of Computing, ACM, 1992, pp. 632–642.

[YK] A. C. YAO AND D. E. KNUTH, *Analysis of the subtractive algorithm for greatest common divisors*, Proc. Nat. Acad. Sci USA 72 (1975), pp. 4720–4722.

[W] B. L. VAN DER WAERDEN, *Gruppen von Linearen Transformationen*, Ergebnisse Math., IV.2, Springer-Verlag, Berlin, 1935.

SIAM J. COMPUT.
Vol. 24, No. 1, pp. 30–45, February 1995

TIGHTER LOWER BOUNDS ON THE EXACT COMPLEXITY OF STRING MATCHING*

RICHARD COLE†, RAMESH HARIHARAN†, MIKE PATERSON‡, AND URI ZWICK§

Abstract. This paper considers the exact number of character comparisons needed to find all occurrences of a pattern of length m in a text of length n using on-line and general algorithms. For on-line algorithms, a lower bound of about $(1 + \frac{9}{4(m+1)}) \cdot n$ character comparisons is obtained. For general algorithms, a lower bound of about $(1 + \frac{2}{m+3}) \cdot n$ character comparisons is obtained. These lower bounds complement an on-line upper bound of about $(1 + \frac{8}{3(m+1)}) \cdot n$ comparisons obtained recently by Cole and Hariharan. The lower bounds are obtained by finding patterns with interesting combinatorial properties. It is also shown that for some patterns off-line algorithms can be more efficient than on-line algorithms.

Key words. string matching, pattern matching, comparisons, complexity, lower bounds

AMS subject classifications. primary 68R15; secondary 68Q25, 68U15

1. Introduction.

The classical *string matching* problem is the problem of finding all occurrences of a pattern $w[1 \ldots m]$ in a text $t[1 \ldots n]$. String matching is among the most extensively studied problems in computer science. A survey of the various algorithms devised for it can be found in [Ah90].

Among the most efficient algorithms devised for string matching are algorithms that gain information about the pattern and text only by performing comparisons between pattern and text characters. Such algorithms need not have any prior knowledge of the (possibly infinite) alphabet from which the pattern and text are drawn. We investigate the exact comparison complexity of string matching in this model and obtain lower bounds on the number of comparisons required (in the worst case). These lower bounds allow the algorithms to preprocess the pattern (but not the text). The lower bounds remain valid even if the algorithms do know the alphabet in advance, provided that the alphabet contains enough characters not appearing in the pattern.

Two kinds of comparison based algorithms have been studied. An *on-line* algorithm is an algorithm that examines text characters only in a window of size m sliding monotonically to the right; furthermore, the window can slide to the right only when all matching pattern instances to the left of the window or aligned with the window have been discovered. A general (or *off-line*) algorithm is an algorithm that can access both the pattern and the text in an unrestricted manner.

Perhaps the most widely known linear time algorithms for string matching are the Knuth–Morris–Pratt [KMP77] and Boyer–Moore [BM77] algorithms. We refer to them as the KMP and BM algorithms, respectively. The KMP algorithm makes at most $2n - m$ comparisons and this bound is tight. The exact complexity of the BM algorithm was an open question until recently. It was shown in [KMP77] that the BM algorithm makes at most $6n$ comparisons if the pattern does not occur in the text. Guibas and Odlyzko [GO80] reduced this to $4n$ under the same assumption. Cole [Cole91] finally proved an essentially tight bound of $3n - \Omega(n/m)$ comparisons for the BM algorithm, whether or not the pattern occurs in the text.

*Received by the editors March 18, 1993; accepted for publication (in revised form) August 30, 1993.

†Courant Institute, New York University, New York, New York 10012. The first two authors were supported in part by National Science Foundation grants CCR-8902221, CCR-8906949, CCR-9202900 and CCR-8901484.

‡Department of Computer Science, University of Warwick, Coventry CV4 7AL, England. This author was supported in part by the ESPRIT BRA Programme of the EC under contracts 3075 (ALCOM) and 7141 (ALCOM II). A part of this work was carried out while this author was visiting Tel Aviv University.

§Department of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. A part of this work was carried out while this author was visiting the University of Warwick.

The versions of the KMP and BM algorithms considered in the preceding paragraph are comparison-based. It is interesting to note that both algorithms have variants that are not purely comparison-based and do not fall into the category of algorithms considered in this paper. The failure function of the KMP algorithm [KMP77] yields finite automata that perform string matching by reading each character exactly once. However, simulations of these automata require prior knowledge of the alphabet and the number of comparisons needed to simulate each transition depends on the alphabet size. Transitions can be simulated in unit time by using text characters to address an array of pointers, but this is not allowed in our model.

The standard BM algorithm [BM77] uses two shift functions to determine the distance to shift the pattern when a mismatch occurs. One of these shift functions, the *occurrence shift*, gives the rightmost position in the pattern in which the unmatched text character occurs. An efficient implementation of this shift function is again alphabet dependent. The second shift function used by the BM algorithm is comparison-based. The analysis of Cole [Cole91] shows that the occurrence shift function does not improve the worst-case behaviour of the BM algorithm. This occurrence shift function is very important in practice, however, as it ensures sublinear time in various probabilistic settings (see [BGR90]). For a study of how the KMP, BM, and other algorithms behave in practice the reader is referred to [HS91].

Apostolico and Crochemore [AC91] gave a simple variant of the KMP algorithm which makes at most $\frac{3}{2}n$ comparisons. Apostolico and Giancarlo [AG86] gave a variant of the BM algorithm which makes at most $2n - m + 1$ comparisons. Crochemore et al. [CCG92] showed recently that remembering just the most recently matched portion reduces the upper bound of BM from $3n$ to $2n$ comparisons.

Recently, Galil and Giancarlo [CGG90], [GG92] analyzed and modified a string matching algorithm designed by Colussi [Coll91] ; they showed that it makes at most $\frac{4}{3}n$ comparisons. In fact, [GG92] give this bound in a sharper form as a function of the period z of the pattern; the bound becomes $n+(n-m)\min\{\frac{1}{3}, \frac{\min\{z,m-z\}+2}{2m}\}$. Galil and Giancarlo [GG91] have also shown that any on-line algorithm for string matching must perform at least $\frac{4}{3}n - O(1)$ comparisons for some strings (the string `aba` is an example). It will be shown here that this lower bound also applies to general algorithms, if only pattern–text comparisons are allowed.

The algorithm of Galil and Giancarlo [GG92] is efficient for relatively short patterns. It may become inefficient for longer patterns. Breslauer and Galil [BG92] and Cole and Hariharan [CH92] have shown that the string matching problem becomes easier as the length of the pattern increases. Breslauer and Galil [BG92] developed an algorithm that performs at most $(1 + O(\frac{\log m}{m})) \cdot n$ character comparisons for texts of length n and patterns of length m. Cole and Hariharan [CH92] obtained an algorithm that performs at most $(1 + O(\frac{1}{m})) \cdot n$ comparisons. As we shall see, this is essentially tight.

Galil and Giancarlo [GG91] showed that any on-line algorithm must perform at least $(1 + \frac{2}{m+3}) \cdot n - O(1)$ comparisons for some patterns of odd length m, and that any (general) algorithm must perform at least $(1+\frac{1}{2m}) \cdot n - O(1)$ comparisons for some patterns of length m.

In this work we improve the lower bounds for both on-line and off-line algorithms. We also show that for certain patterns off-line algorithms can be more efficient than on-line algorithms. Some of our lower bounds apply in a model in which both text–text and pattern–text comparisons are allowed. We suspect that for some patterns text–text comparisons can improve the efficiency of string matching algorithms.

Our improved lower bounds are the following: For on-line algorithms that use only pattern–text comparisons, a lower bound of $(1 + \frac{16}{7m+27}) \cdot n - O(1)$ character comparisons is obtained, for $m = 16k + 19$ where $k \geq 1$. For on-line algorithms that are allowed to use both pattern–text and text–text comparisons, a lower bound of $(1 + \frac{9}{4(m+1)}) \cdot n - O(1)$ character comparisons is obtained for $m = 36k + 35$, where $k \geq 0$. For general off-line

algorithms that are allowed to use both pattern–text and text-text comparisons, a lower bound of $(1 + \frac{2}{m+3}) \cdot n - O(1)$ character comparisons is obtained for $m = 2k + 1$, where $k \geq 2$. We also get an off-line lower bound of $\frac{4}{3}n - O(1)$ character comparisons for $m = 3$ if only pattern–text comparisons are allowed.

The on-line lower bounds presented come very close to the on-line upper bound of $(1 + \frac{8}{3(m+1)}) \cdot n$ obtained by Cole and Hariharan [CH92]. The worst-case comparison complexity of string matching is therefore almost exactly determined. It is asymptotically of the form $(1 + \frac{d}{m}) \cdot n$ where for on-line algorithms $\frac{9}{4} \leq d \leq \frac{8}{3}$ and for general algorithms $2 \leq d \leq \frac{8}{3}$.

Our work builds on the work of Galil and Giancarlo [GG91]. Our point of view, however, is a bit different. Galil and Giancarlo [GG91] investigated the number of comparisons required only as a function of n, the text length, and m, the pattern length. We are interested in the number of comparisons required as a function of the text length and the specific pattern sought.

In the next section we explain, in more detail, the rules of the string matching game in the comparison model setting. In §3 we describe the adversary arguments that lie at the heart of our lower bounds proofs. The off-line lower bounds presented in §4 follow almost immediately from the arguments of §3. A specific lower bound is obtained for every pattern. This lower bound depends on the first and second periods of the pattern (see next section). These off-line lower bounds are shown to be tight for an interesting family of patterns. Exploiting the additional restrictions for on-line algorithms, we obtain, in §§5 and 6, improved on-line lower bounds. The lower bound of §5 depends again on the first and second periods of the patterns. Additional periods and more complicated combinatorial structures are used in §6. In §7 we obtain some on-line upper bounds (for strings of the form $\mathtt{a}^k\mathtt{ba}^\ell$) that match the on-line and some off-line lower bounds of §§4 and 5. Finally, in §8 we exhibit a pattern (abaa) for which an off-line algorithm (it is actually on-line with a small look-ahead) performs better than any on-line algorithm.

A preliminary version of this paper has appeared in [CHPZ93].

2. Preliminaries. The algorithms we consider are allowed to access the text and the pattern only through queries of the form "$t[i] = w[j]$?" or "$t[i] = t[j]$?". To each such query the algorithm is supplied with a "yes" or "no" answer. An algorithm is charged only for the queries it makes; all other computations are free of charge. Algorithms may adaptively choose their queries depending on the answers to earlier queries. An algorithm in this model may be viewed as a sequence of decision trees. Similar comparison models are used to study comparison problems such as sorting, searching, and selection.

For a string w, let $c(w)$ denote the minimal constant for which there exists a string matching algorithm that finds all occurrences of the pattern w in a text of length n using at most $c(w) \cdot n + o(n)$ comparisons (between text and pattern characters and between pairs of text characters). A variant of $c(w)$ is $c^*(w)$ in which the algorithm is not allowed to compare pairs of text characters. Obviously, $c(w) \leq c^*(w)$.

In the definition of $c(w)$ and $c^*(w)$, we allow unrestricted *off-line* algorithms that have random access to all the characters of the text. By contrast, we define $c_k(w)$ and $c_k^*(w)$ to be the corresponding minimal constants when the algorithms have access to the text only through a sliding window of size $|w|+k$ (where $|w|$ denotes the length of w). Furthermore, the algorithm is only allowed to slide the window past a text position when it has already reported whether an occurrence of the pattern starts at that text position. Algorithms using a sliding window of size $|w|$ (i.e., $k = 0$) are traditionally called *on-line* algorithms. We call algorithms that use larger windows *finite look-ahead* or *window* algorithms. Clearly $c(w) \leq c_k(w) \leq c_0(w)$ for any $k \geq 0$. We show in §8 that for some w and some k, $c_k(w) < c_0(w)$. More specifically, we show there that $c_4(\mathtt{abaa}) < c_0(\mathtt{abaa})$. This means that for some patterns, algorithms that use larger windows may be more efficient than all algorithms that use smaller windows. It is

still an open problem whether there exists a string w for which $c(w) < c_k(w)$ for every $k \geq 0$. That is, it is not known whether there exists strings for which an optimal off-line algorithm is better than any finite look-ahead algorithm. It is clear however that $c_k(w)$ is nonincreasing in k. The following lemma is also easily established.

LEMMA 2.1. *For any string w we have* $\lim_{k\to\infty} c_k(w) = c(w)$.

Proof. Let $c(n)$ be the number of comparisons required, in the worst case, to find all occurrences of w in a text of length n using an unrestricted algorithm. By the definition of $c(w)$ we get that $c(n) \leq c(w)\cdot n + d(n)$ where $d(n) = o(n)$. For every $k \geq 0$ consider now the following algorithm with look-ahead k. The algorithm finds all occurrences of w in its window of size $k + |w|$ using at most $c(k + |w|)$ comparisons. The window is then slid by $k + 1$ positions and the same process is repeated. The number of comparisons performed by this algorithm on a text of length n is at most

$$\left\lceil \frac{n}{k+1} \right\rceil \cdot c(k + |w|) \leq \hat{c}_k(w)\cdot n + \hat{d}_k(w),$$

where

$$\hat{c}_k(w) = \frac{k + |w|}{k+1} \cdot c(w) + \frac{d(k + |w|)}{k+1}$$

and $\hat{d}_k(w)$ is some constant (depending on w and k). In particular we get that $c_k(w) \leq \hat{c}_k(w)$. It is now easy to check that $\lim_{k\to\infty} \hat{c}_k(w) = c(w)$ and therefore $\lim_{k\to\infty} c_k(w) \leq c(w)$. It is clear however that $\lim_{k\to\infty} c_k(w) \geq c(w)$ and the required equality follows. □

It is easy to see that $1 \leq c(w) \leq c_0(w)$, $c^*(w) \leq c_0^*(w)$ for every string w. The KMP algorithm shows that $1 \leq c_0^*(w) \leq 2$, for every w. The algorithm of Galil and Giancarlo [GG92] shows that $1 \leq c_0^*(w) \leq \frac{4}{3}$, for every w. The algorithm of Breslauer and Galil [BG92] shows that $1 \leq c_0^*(w) \leq 1 + \frac{4\log_2 m+2}{m}$ for every string of length m. Finally, the algorithm of Cole and Hariharan [CH92] shows that $1 \leq c_0^*(w) \leq 1 + \frac{8}{3(m+1)}$ for every string w of length m. The algorithms (of [KMP77],[GG92],[BG92], and [CH92]) mentioned here are all on-line and they use only pattern–text comparisons.

Galil and Giancarlo [GG91] showed that $c_0^*(w) \geq c_0(w) \geq 1 + \frac{2}{m+3}$ for some patterns of odd length m. We show that for infinitely many values of m there exists strings of length m for which $c_0^*(w) \geq c_0(w) \geq 1 + \frac{9}{4(m+1)}$. We also show that for infinitely many values of m there exists strings of length m for which $c_0^*(w) \geq 1 + \frac{16}{7m+27}$. This shows that the algorithm of Cole and Hariharan is not far from being optimal. We further show that $c^*(w) \geq c(w) \geq 1 + \frac{2}{m+3}$ for some patterns of odd length $m \geq 5$, showing essentially that the lower bounds obtained by [GG91] for on-line algorithms also hold for general algorithms.

Let w be a string of length m. We say that z ($1 \leq z \leq m$) is *a period* of w if and only if $w[i] = w[i + z]$ for every $1 \leq i \leq m - z$. Let z_1 be the minimal period of w. (A minimal period exists since m is always a period of w.) Let z_2 be the minimal period of w which is not divisible by z_1. If such a second period does not exist we set $z_2 = \infty$. We call z_1 *the period* of w and z_2 *the second period* of w. Periodicity properties play a major role in the sequel.

It is well known (see, e.g., [KMP77]) that if z_1 and z_2 are periods of w and if $z_1 + z_2 \leq |w| + \gcd(z_1, z_2)$ then $\gcd(z_1, z_2)$ is also a period of w. If z_1 and z_2 are the first and second periods of w then $\gcd(z_1, z_2)$ is not a period of w and, as a consequence, $z_1 + z_2 \geq |w| + 2$.

3. Adversary arguments. Our lower bounds are derived using an adversary that fills in the text while answering the algorithm's queries. The adversary always "tiles" the text with (overlapping) occurrences of the pattern. Every character of the text eventually becomes part of an occurrence, which the algorithm must find. Consequently the algorithm must establish

the identity of each text character and it can achieve this only by getting at least one "yes" answer for each position. The adversary tries to avoid giving "yes" answers whenever possible. It gives a "yes" answer only when a "no" answer would either contradict a previous answer or prevent it from completely tiling the text. The arguments of this section are generalizations of similar arguments of Galil and Giancarlo [GG91].

The previous statement, that at least one "yes" answer must be obtained by the algorithm for each text position covered by an occurrence of the pattern, seems obvious. It is indeed immediate if only pattern–text comparisons are allowed. A slightly more complicated argument is needed to handle the possibility of text–text comparisons.

LEMMA 3.1. *A comparison based algorithm can be certain about the identity of s text characters in a text t only after receiving at least s "yes" answers.*

Proof. We construct a graph G which has one vertex for each text position and one vertex for each of the k distinct symbols which appear in the pattern w. Every edge in G corresponds to a "yes" answer received by the algorithm. If a "yes" answer was given to a query "$w[i] = t[j]$?" then an edge is added between the vertex corresponding to $t[j]$ and the vertex corresponding to the symbol at $w[i]$. If a "yes" answer was given to a query "$t[i] = t[j]$?" then an edge is added between the vertices corresponding to $t[i]$ and $t[j]$.

At any stage of the algorithm the graph G constructed so far represents the positive information known about the characters in the text. The text positions corresponding to vertices in components of G which contain a pattern symbol vertex are the only text positions where the identity of the character is known. Since the alphabet size is unlimited, the character at any other text position is not yet determined. Since a component of size p containing a pattern symbol vertex has $p - 1$ text vertices and at least $p - 1$ edges, the total numbers of known text positions is at most the total number of "yes" answers. □

Next we describe a scheme using which the adversary can give any algorithm a relatively large number of "no" answers. We begin with a simple example.

The pattern string is aba, and let $n = 3r + 1$ for some $r \geq 1$. We consider the family $\mathcal{F} = \{t_v : v \in \{0, 1\}^r\}$ of text strings of length n defined as follows. Place a's in positions $3j$, for $0 \leq j < r$, of all texts t_v. In positions $3j + 1, 3j + 2$ of t_v, put ba if $v_j = 0$ and ab if $v_j = 1$. (For simplicity, we number the positions here from 0.) This family may be depicted schematically as

$$\ldots \mathtt{a} \begin{matrix}\mathtt{ba}\\ \mathtt{ab}\end{matrix} \mathtt{a} \begin{matrix}\mathtt{ba}\\ \mathtt{ab}\end{matrix} \mathtt{a} \begin{matrix}\mathtt{ba}\\ \mathtt{ab}\end{matrix} \mathtt{a} \ldots .$$

Finding all occurrences of aba in a text string from $\mathcal{F}$ is equivalent to determining the index vector $v \in \{0, 1\}^r$. An adversary can force at least one "no" answer before revealing each bit of v.

The following definition generalises the properties of the example.

DEFINITION 3.2. Let w be a string. A family $\mathcal{F} = \{t_v : v \in \{0, 1\}^r\}$ of texts of length n is said to be *r-separating* for w if there exist distinct indices $u_1^0, \ldots, u_r^0, u_1^1, \ldots, u_r^1$ such that the following holds:

1. For every $v = (v_1, \ldots, v_r) \in \{0, 1\}^r$ and for every i, the text t_v contains an occurrence of w starting at position u_i^0 if and only if $v_i = 0$ and an occurrence of w at position u_i^1 if and only if $v_i = 1$.

2. The answer to any query of the form "$w[i] = t[j]$?" is either "yes" for all texts t_v, or "no" for all texts t_v, or "yes" for a text t_v if and only if $v_k = \varepsilon$, for some fixed $1 \leq k \leq r$ and $\varepsilon \in \{0, 1\}$.

3. The answer to a text–text query "$t[i] = t[j]$?" is either "yes" for all texts t_v, or "no" for all t_v, or "yes" for t_v if and only if $v_{k_1} = \varepsilon_1$, or "yes" for t_v if and only if $v_{k_1} \oplus v_{k_2} = \varepsilon_1$, or "yes" in t_v if and only if $v_{k_1} = \varepsilon_1$ and $v_{k_2} = \varepsilon_2$, for some fixed $1 \leq k_1, k_2 \leq r$ and $\varepsilon_1, \varepsilon_2 \in \{0, 1\}$. (If $\mathcal{F}$ contains only symbols from the pattern then 3. follows from 1. and 2.)

In the example given before Definition 3.2, there is no text–text query whose answer is "yes" if and only if $v_{k_1} = \varepsilon_1$ and $v_{k_2} = \varepsilon_2$, for some fixed $1 \leq k_1 \neq k_2 \leq r$ and $\varepsilon_1, \varepsilon_2 \in \{0, 1\}$. Such a situation may arise however for patterns w that contain more than two distinct characters.

We are now ready to prove the next lemma.

LEMMA 3.3. *If $\mathcal{F}$ is an r-separating family for w then, for any comparison-based algorithm for w, there exists a text $t_v \in \mathcal{F}$ for which the algorithm receives at least r "no" answers before being able to locate all the occurrences of w in t_v.*

Proof. The adversary maintains a set E containing linear equations over the binary field GF(2) in the variables $v_1, \ldots, v_r$. At any stage, there is at least one vector $v \in \{0, 1\}^r$ that satisfies all the equations of E, and if a vector $v \in \{0, 1\}^r$ satisfies all the equations of E then the text t_v is consistent with all answers given so far by the adversary. Further, the number of equations in E is at most the number of "no" answers given by the adversary. At the beginning $E = \phi$, and as no query has been made, all texts are still possible. This is how the adversary responds to a new query:

If the answer to the query is the same for all texts t_v for which v is a solution of E, the adversary responds with this common answer. The set E remains unchanged and all the invariants remain satisfied.

Otherwise, the adversary answers with a "no". It then adds an equation to E in the following way. As the answer to the current query is not the same for all the texts in $\mathcal{F}$, there exist, by Definition 3.2, either a single equation e_1 or two equations e_1 and e_2 such that the answer to the query is "yes" in t_v if and only if v satisfies e_1 or both e_1 and e_2.

If the answer to the query, according to t_v, is "yes" if and only if e_1 is satisfied, then $\overline{e_1}$, the equation obtained from e_1 by complementing its free coefficient, is added to E. If the answer to the query, according to t_v, is "yes" if and only if both e_1 and e_2 are satisfied, then at least one of e_1 and e_2 is independent of the equations of E, as otherwise the answer would have been the same for all surviving texts. If e_1 does not depend on E then the equation $\overline{e_1}$ is added to E, otherwise $\overline{e_2}$ is added. It is easy to verify that all the required invariants are still satisfied.

The algorithm's task is done only when there is a unique solution to E. This happens only when the set E contains at least r equations. An equation is added to E only as a result of a "no" answer. The adversary can therefore give the algorithm at least r "no" answers. □

Lemmas 3.1 and 3.3 can be combined together to give a lower bound of $n + r$ if, in every text t_v of the separating family $\mathcal{F}$ used in Lemma 3.3, every text position is covered by an occurrence of the pattern w. Such separating families will be constructed in the next section.

4. Off-line lower bounds.

THEOREM 4.1. *If w is a string and z_1, z_2 are its first and second periods then $c(w) \geq 1 + \frac{1}{z_1 + z_2}$.*

Proof. Assume without loss of generality that $n = r(z_1 + z_2) + |w|$ for some $r \geq 1$. For every $v \in \{0, 1\}^r$ construct a text t_v of length n in which, for every $0 \leq j < r$, occurrences of w start at $j(z_1 + z_2)$, and either at $j(z_1 + z_2) + z_1$ or $j(z_1 + z_2) + z_2$ according to whether $v_j = 0$ or $v_j = 1$. It is easy to verify that $\mathcal{F} = \{t_v : v \in \{0, 1\}^r\}$ is an r-separating family for w where $u_j^0 = j(z_1 + z_2) + z_1$ and $u_j^1 = j(z_1 + z_2) + z_2$, for $0 \leq j < r$. This construction is depicted in Fig. 1 (note that $z_1 + z_2 \geq |w| + 2$). Consider now a comparison-based algorithm A that finds all occurrences of w in a string of length n. According to Lemma 3.3, A gets at

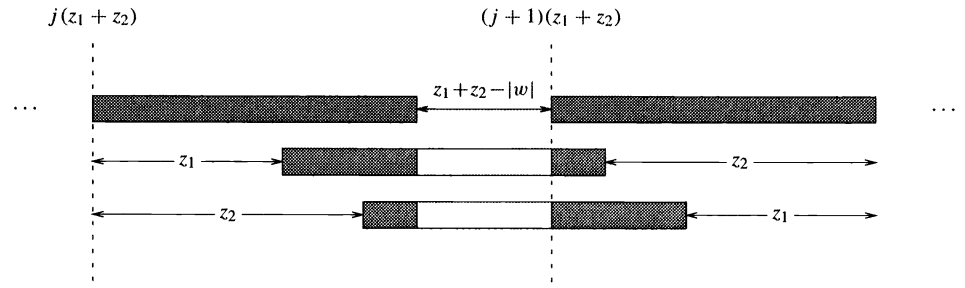


FIG. 1. *The configuration used to prove that* $c(w) \geq 1 + \frac{1}{z_1+z_2}$.

least r "no" answers for at least one text t_{v_0}, where $v_0 \in \{0,1\}^r$. It is also easy to see that every text t_v, and in particular t_{v_0}, is completely covered with occurrences of w. According to Lemma 3.1, A must therefore get at least n "yes" answers on t_{v_0}. In total, A must make, in the worst case, at least $n+r = (1+\frac{1}{z_1+z_2})n - \frac{|w|}{z_1+z_2}$ comparisons for a text of length n. □

As an example, note that for the string aba, $z_1 = 2$ and $z_2 = 3$ and therefore $c(\texttt{aba}) \geq \frac{6}{5}$. The separating family used to obtain this lower bound may be depicted as

$$\ldots \texttt{aba} \begin{matrix}\texttt{ba}\\ \texttt{ab}\end{matrix} \texttt{aba} \begin{matrix}\texttt{ba}\\ \texttt{ab}\end{matrix} \texttt{aba} \begin{matrix}\texttt{ba}\\ \texttt{ab}\end{matrix} \texttt{aba} \ldots .$$

This family has the property that, in every text of the family, every position is covered by an occurrence of aba. The separating family for aba given after Definition 3.2 did not have this property.

As a further example, note that for the string abaa we have $z_1 = 3$ and $z_2 = 4$, and therefore $c(\texttt{abaa}) \geq \frac{8}{7}$. In §8 it will be shown that this bound is tight, i.e., $c(\texttt{abaa}) = \frac{8}{7}$. We will see from Theorem 7.1 that $c_0(\texttt{abaa}) = \frac{5}{4}$. This provides an example of a string for which off-line algorithms can be more efficient than on-line algorithms.

THEOREM 4.2. *If w is a string, z_1, z_2 are its first and second periods, and $2z_2 - z_1 \leq |w|$ then $c(w) \geq 1 + \frac{1}{z_2}$.*

Proof. The proof is very similar to the proof of Theorem 4.1. A separating family, in which every text is *almost* completely tiled with occurrences of w, may be obtained this time without using occurrences of w that are common to all the texts of the family.

Assume that $n = rz_2 + |w| + z_1$ for some $r \geq 1$. For every $v \in \{0,1\}^r$, construct a text t_v of length n in which, for $0 \leq j < r$, occurrences of w start at jz_2 if $v_j = 0$ or at $jz_2 + (z_2 - z_1)$ if $v_j = 1$. It is again easy to check that $\mathcal{F} = \{t_v : v \in \{0,1\}^r\}$ is an r-separating family for w where this time $u_j^0 = jz_2$ and $u_j^1 = jz_2 + (z_2 - z_1)$, for every $0 \leq j < r$. The construction is depicted in Fig. 2. Note that if z_1, z_2 ($z_1 < z_2$) are periods of w then so is $2z_2 - z_1$. As $2z_2 - z_1 \leq |w|$, every position in a text t_v, except perhaps the first and last $z_2 - z_1$ positions, is covered by an occurrence of w. Thus, as in the proof of Theorem 4.1, we can show that any algorithm must perform, in the worst case, at least $n(1 + \frac{1}{z_2}) - O(|w|)$ comparisons. □

As an example, for the string aabaa we have $z_1 = 3$, $z_2 = 4$ and $2z_2 - z_1 \leq |w|$ and therefore $c(\texttt{aabaa}) \geq \frac{5}{4}$. The separating family used this time is

$$\ldots \texttt{aa} \begin{matrix}\texttt{ba}\\ \texttt{ab}\end{matrix} \texttt{aa} \begin{matrix}\texttt{ba}\\ \texttt{ab}\end{matrix} \texttt{aa} \begin{matrix}\texttt{ba}\\ \texttt{ab}\end{matrix} \texttt{aa} \ldots .$$

THEOREM 4.3. $c^*(\texttt{aba}) = \frac{4}{3}$.

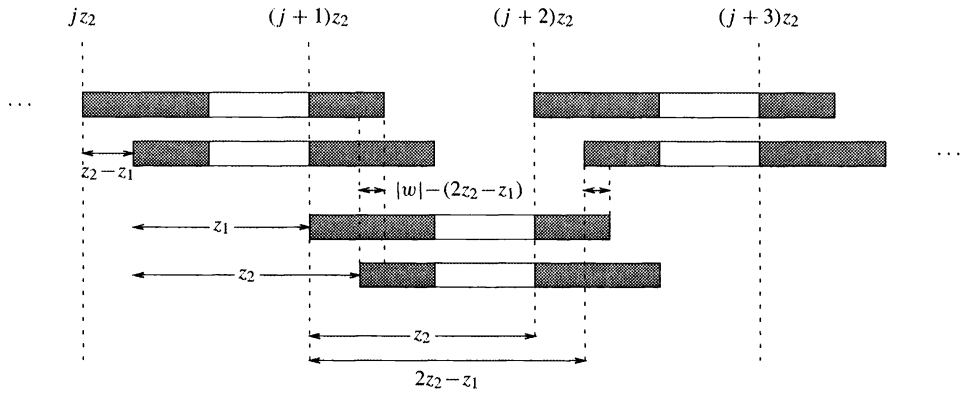


FIG. 2. *A configuration used to show that* $c(w) \geq 1 + \frac{1}{z_2}$.

Proof. The upper bound will follow from Theorem 7.1. The lower bound does not follow from Theorem 4.2 as the condition $2z_2 - z_1 \leq |w|$ is not satisfied. A specialized argument is needed in this case. The argument given here assumes that only pattern–text comparisons are allowed. It does not extend in a simple manner to the case in which both pattern–text and text–text comparisons are allowed.

The lower bound is obtained using the separating family for aba given after Definition 3.2. A complication arises, however, as texts in this family are not completely covered by occurrences of aba.

Assume that $n = 3r + 1$ for some $r \geq 1$. The adversary starts by putting a's in all text positions $3j$ for $0 \leq j < r$. It will set positions $3j + 1, 3j + 2$ to either ab or ba only after replying with a "no" to at least one query concerning these positions.

The adversary answers the queries of the algorithm in the following way. If the queried text position was already set by the adversary, the answer consistent with this setting is returned. If the query is "$t[3j + k] = \mathtt{a}$?" or "$t[3j + k] = \mathtt{b}$?" where $k = 1, 2$, and position $3j + k$ has not yet been set, the adversary responds with a "no". It then sets positions $3j + 1, 3j + 2$ to either ab or ba, whichever is consistent with its "no" answer.

All text positions of the form $3j + 1$ and $3j + 2$ will eventually be covered by occurrences of aba. The adversary therefore forces at least one "no" answer and two "yes" answers for each such pair. Positions of the form $3j$ are not necessarily covered by occurrences of aba. If, however, position $3j$ is not covered by such an occurrence, then positions $3j - 2, 3j - 1$ are set to ba and positions $3j + 1, 3j + 2$ are set to ab. An algorithm must still query position $3j$ at least once in such a case, to either verify or rule out an occurrence of aba starting at position $3j - 1$. This completes the proof. □

For a nonperiodic string w (i.e., a string with $z_1 = |w|$, $z_2 = \infty$), the above theorems give only the trivial lower bound, $c(w) \geq 1$. This bound is tight, however, as the many string matching algorithms (see, e.g., those of [Coll91],[GG91], and [CH92]) perform at most n comparisons when searching for a nonperiodic pattern in a text of length n.

As a corollary to Theorem 4.2 we get the following corollary.

COROLLARY 4.4. *For* $k, \ell \geq 2$ *we have* $c(\mathtt{a}^k\mathtt{ba}^\ell) \geq 1 + \frac{1}{\max\{k,\ell\}+2}$.

Proof. It is easy to check that the first and second periods of $w = \mathtt{a}^k\mathtt{ba}^\ell$ are $z_1 = \max\{k, \ell\} + 1$ and $z_2 = \max\{k, \ell\} + 2$ and that $2z_2 - z_1 = \max\{k, \ell\} + 3 \leq |w| = k + \ell + 1$. The claim follows immediately from Theorem 4.2. □

In §7 it will be shown that the bounds given in Corollary 4.4 are tight. They can even be matched using on-line algorithms. As a further corollary to Theorem 4.2 (or Corollary 4.4) we get the following corollary.

COROLLARY 4.5. *For every $m = 2k+1$, where $k \geq 2$, there exists a string $w_m (= \mathtt{a}^k\mathtt{b}\mathtt{a}^k)$ of length m such that any algorithm that finds all the occurrences of w_m in a text of length n must make at least $(1 + \frac{2}{m+3})n - O(1)$ comparisons in the worst case.*

We know (see the last paragraph of §2) that if z_1 and z_2 are the first and second periods of w then $z_1 + z_2 \geq |w| + 2$. As $z_2 \geq z_1 + 1$, we get that $z_2 \geq \lceil \frac{|w|+3}{2} \rceil$. Corollary 4.5 is therefore the strongest result of its kind implied by Theorem 4.2.

5. On-line lower bounds I. In this short section we show that the lower bound, $c(w) \geq 1 + \frac{1}{z_2}$, obtained for off-line algorithms only when $2z_2 - z_1 \leq |w|$, holds for on-line algorithms even if this condition does not hold.

THEOREM 5.1. *If w is a string and z_2 is its second period then $c_0(w) \geq 1 + \frac{1}{z_2}$.*

Proof. Suppose that an on-line algorithm has just found an occurrence of w in the text. The window will now be slid by at most z_1 positions to the right. Place two copies of w shifted by z_1 and z_2 positions below w, as shown in Fig. 3. Denote these copies by w' and w''. Since $z_2 - z_1$ is not a period of w, the two copies w' and w'' must disagree in at least one position after the end of the found occurrence of w. The adversary will extend the found occurrence of w by either w' or w'' in a way that will force the algorithm to get at least one "no" answer. If the algorithm makes a query whose answer is identical under both continuations, the adversary gives the algorithm this common answer. At some stage the algorithm has to make a query that distinguishes between the two incompatible continuations. No matter what this query is, the adversary answers it by "no". The adversary now chooses the continuation consistent with this "no" and answers all further questions accordingly, until the algorithm finds the chosen occurrence. By then the algorithm has either made at least $z_1 + 1$ queries and can slide the window by only z_1 positions, or has made at least $z_2 + 1$ queries and can slide the window by only z_2 positions. Note that to verify an occurrence of the pattern in the text, the algorithm must get at least one "yes" answer for each character of this occurrence. This process will be repeated again and again, forcing the algorithm to make at least $(1 + \frac{1}{z_2}) \cdot n - O(1)$ queries on a text of length n. □

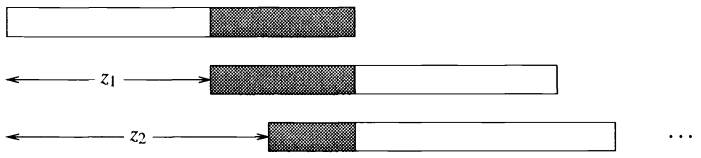


FIG. 3. *The configuration used to prove that $c_0(w) \geq 1 + \frac{1}{z_2}$.*

In the next section we obtain, using more complicated arguments, better lower bounds for on-line algorithms (see Corollaries 6.3 and 6.5).

6. On-line lower bounds II. In (the proof of) Theorem 5.1 it was shown that for every nonperiodic pattern the adversary can force any algorithm to make at least *one* mistake (i.e., get at least one "no" answer) for each occurrence of the pattern used in the tiling of the text. Now we show that for certain patterns the adversary can force any algorithm to make at least *two* mistakes for each such occurrence. The algorithm of Cole and Hariharan [CH92] makes at most two mistakes for each such occurrence, so no adversary can force all algorithms to make at least three mistakes for each occurrence of the tiling.

THEOREM 6.1. *Let w be a string of length m and let $z_1 < z_2 < \cdots < z_k$ be periods of w such that for every $1 \leq i < j \leq k$, $z_j - z_i$ is not a period of w.*

(i) *If none of the multisets $\{w[m + i - z_j] : 1 \leq j \leq k\}$ for $1 \leq i \leq z_1$ contains a character exactly $k - 1$ times, then $c_0^*(w) \geq 1 + \frac{2}{z_k}$.*

(ii) *If, in addition, none of the multisets* $\{(w[m+i_1-z_j], w[m+i_2-z_j]) : 1 \le j \le k\}$ *for* $1 \le i_1 < i_2 \le z_1$ *contains exactly* $k-1$ *equal pairs, then* $c_0(w) \ge 1 + \frac{2}{z_k}$.

Before proceeding with the proof of this theorem, we try to clarify the conditions appearing in it. Consider $k+1$ copies of w, positioned in an array of $k+1$ rows numbered $0, 1, \ldots, k$, and $m + z_k$ columns numbered $1, \ldots, m + z_k$, where the copy in the ith row is shifted z_i positions to the right with respect to the copy in the 0th row. Such arrays are depicted in Fig. 4(a) for the string $w_{10} = 1213451121$ with $z_1 = 7$, $z_2 = 9$, and $z_3 = 10$, and in Fig. 4(b) for the string $w_{12} = 121342531121$ with $z_1 = 9$, $z_2 = 11$, and $z_3 = 12$. A multiset $\{w[m+i-z_j] : 1 \le j \le k\}$ contains the k characters appearing in column $m+i$ of rows $1, \ldots, k$ in the array corresponding to w. The requirement in clause (i) above is that in each column that lies after the end of the copy of the 0th row, but at or before the end of all the other copies, no character appears in all but one of the rows. It is easy to check that in both cases depicted in Fig. 4 this condition is satisfied. Note that when $k = 3$ this condition requires that the three characters in such a column will either all be equal or all be distinct.

```
1213451121                  121342531121
      1213451121                  121342531121
        1213451121                  121342531121
         1213451121                  121342531121
```

(a) The set up $\langle w_{10}; 7, 9, 10\rangle$ (b) The set up $\langle w_{12}; 9, 11, 12\rangle$

FIG. 4. *Two simple setups in which Theorem* 6.1 *can be applied.*

To check the condition of clause (ii) above, one needs to look at pairs of such columns and compare the pair of characters appearing in each row. The number j of equal pairs is required to satisfy $j \ne k-1$. It is easily verified that this condition is satisfied in the array of $w_{10} = 1213451121$ but not in the array of $w_{12} = 121342531121$. Thus for w_{10} we obtain $c_0(w_{10}) \ge \frac{6}{5}$, while for w_{12} we can only infer $c_0^*(w_{12}) \ge \frac{7}{6}$.

Proof. The proof (of both statements) is a simple extension of the proof of Theorem 5.1. Suppose that an on-line algorithm has just found an occurrence of w in the text. The window can be slid at most z_1 positions to the right. Below w, place k copies of w shifted by $z_1, z_2, \ldots, z_k$ positions, respectively. (The reader may refer to Fig. 3 imagining that k instead of just two copies appear there.) Since none of $z_j - z_i$ is a period of w, each pair of copies must disagree in at least one position after the end of the found occurrence of w. The adversary will extend the found occurrence of w by one of the k copies in a way that will force the algorithm to get at least two "no" answers. If the algorithm asks a question whose answer under the above k continuations is the same, the adversary gives the algorithm this common answer. At some stage the algorithm has to make a query to which the answer is "yes" according to some of the continuations, and "no" according to the rest of them. The adversary will answer this query with a "no". Conditions (i) and (ii) imply that at least two continuations are consistent with this "no" answer. The adversary now gives the common answers to all queries that do not distinguish between the remaining continuations. At some stage the algorithm has to make another query to which both answers are possible. Again, the adversary answers this with a "no". At least one continuation is consistent with all the replies given by the adversary. The adversary chooses one of them and answers all subsequent queries accordingly, until the algorithm finds the next occurrence of w. By then the algorithm has made at least $z_i + 2$ queries, for some $1 \le i \le k$, and it can slide the window by only z_i positions. □

We will henceforth say that $\langle w; z_1, z_2, \ldots, z_k\rangle$ is a *setup* if w is a string, and $z_1 < z_2 < \cdots < z_k$ are periods of w, and none of $z_j - z_i$, for $i \neq j$, is a period of w. The string w_{10} is the shortest string for which a setup satisfying the conditions of Theorem 6.1 can be obtained. The string w_{12} is the shortest string for which a setup that satisfies condition (i), but not condition (ii), of Theorem 6.1 can be obtained. The two last statements were verified using a computer search.

We next show how to obtain from each setup satisfying the conditions of Theorem 6.1 an infinite sequence of such setups. This helps in the investigation of the asymptotic number of comparisons required as the length of the pattern strings tends to infinity. The infinite sequence is obtained by *padding* the basic setup.

Let u and v be strings. $\mathrm{pad}(u, v)$ denotes the string obtained by placing a copy of v before and after each character of u. Thus $\mathrm{pad}(121, 00) = 00100200100$ and in general $|\mathrm{pad}(u, v)| = (|u| + 1)(|v| + 1) - 1$.

THEOREM 6.2. *If $\langle w; z_1, \ldots, z_k\rangle$ is a setup satisfying the conditions of Theorem* 6.1 *and if $w_\ell = \mathrm{pad}(w, 0^\ell)$, then*

$$c_0(w_\ell) \geq 1 + \frac{2(|w|+1)}{z_k} \cdot \frac{1}{|w_\ell|+1} \quad .$$

Proof. If the setup $\langle w; z_1, \ldots, z_k\rangle$ satisfies the conditions of Theorem 6.1 then so does the setup $\langle w_\ell; (\ell+1)z_1, \ldots, (\ell+1)z_k\rangle$. To see this, note at first that $(\ell+1)z_i$ for $1 \leq i \leq k$ is indeed a period of $w_\ell = \mathrm{pad}(w, 0^\ell)$ and that none of $(\ell+1)(z_j - z_i)$ for $i \neq j$ is such a period. To verify the first condition of Theorem 6.1, note that every column in the setup $\langle w_\ell; (\ell+1)z_1, \ldots, (\ell+1)z_k\rangle$ is either a column of the setup $\langle w; z_1, \ldots, z_k\rangle$ or an all-zero column. The second condition is verified in a similar way. The statement of the theorem then follows from Theorem 6.1, applied to $\langle w_\ell; (\ell+1)z_1, \ldots, (\ell+1)z_k\rangle$, and from the fact that $1 + \frac{2}{(\ell+1)z_k} = 1 + \frac{2(|w|+1)}{z_k} \cdot \frac{1}{|w_\ell|+1}$. □

Theorem 6.2 motivates the search for setups $\langle w; z_1, \ldots, z_k\rangle$ satisfying the conditions of Theorem 6.1 for which $2(|w| + 1)/z_k$ is as high as possible. The best such setup that we have found with $k = 3$ is the following $\langle w_{35}; 25, 30, 32\rangle$:

```
12121121213412156781479121212112121
                        12121121213412156781479121212112121
                            12121121213412156781479121212112121
                              12121121213412156781479121212112121
```

For this setup, $2(|w| + 1)/z_k = \frac{9}{4}$. As a corollary to Theorem 6.2 we obtain the next corollary.

COROLLARY 6.3. *For every $m = 36k + 35$, where $k \geq 0$, there exists a string w_m of length m such that any on-line algorithm that finds all the occurrences of w_m in a text of length n must make at least $(1 + \frac{9}{4(m+1)})n - O(1)$ comparisons in the worst case.*

Using a computer enumeration we have verified that no better setup with $k = 3$ is possible with a pattern of length at most 250. However, better setups that satisfy the first condition of Theorem 6.1 can be obtained by using four instead of three overlaps.

Let

$$u_k = \left(\left((12)^k\,1\right)^2 3^3\right)^2 12 \left((12)^k\,1\right)^2 3^3 \left((12)^k\,1\right)^2 \quad .$$

The following lemma is easily verified.

LEMMA 6.4. *The setups $\langle u_k; 8k + 12, 12k + 17, 14k + 18, 14k + 20\rangle$ for $k \geq 1$ satisfy the first condition of Theorem* 6.1.

The setup $\langle u_1; 20, 29, 32, 34\rangle$ for example is

```
12112133312112133312121121333121121
                  12112133312112133312121121333121121
                          12112133312112133312121121333121121
                             12112133312112133312121121333121121
                               12112133312112133312121121333121121
```

As $|u_k| = 16k + 19$, we get as a corollary the following.

COROLLARY 6.5. *For every $m = 16k+19$, where $k \geq 1$, there exists a string $w_m (= u_k)$ of length m such that any on-line algorithm that uses only pattern–text comparisons to find all the occurrences of w_m in a text of length n must make at least $(1 + \frac{16}{7m+27})n - O(1)$ comparisons in the worst case.*

Corollary 6.5 is asymptotically better than Corollary 6.3 and it is the best on-line bound we have obtained. We have verified using a computer search that no better setup with four or five overlaps can be obtained using strings of length at most 250.

We believe that if $\langle w; z_1, \ldots, z_k \rangle$ is a setup satisfying the conditions of Theorem 6.1 then $|z_k| \geq \frac{7}{8}|w|$. If this is true, then the result of Corollary 6.5 is essentially the best that can be obtained using our methods.

7. On-line upper bounds. The next theorem exhibits an interesting family of strings for which Theorem 5.1 is tight.

THEOREM 7.1. *For every $k, \ell \geq 1$ we have $c_0(\mathtt{a}^k\mathtt{b}\mathtt{a}^\ell) = c_0^*(\mathtt{a}^k\mathtt{b}\mathtt{a}^\ell) = 1 + \frac{1}{\max\{k,\ell\}+2}$.*

Proof. The lower bound is a corollary of Theorem 5.1. A matching upper bound is fairly straightforward for the case $k \leq \ell$, but needs more care when $k > \ell$. We will describe an algorithm that works in both cases.

Algorithm for $\mathtt{a}^k\mathtt{b}\mathtt{a}^\ell$. The algorithm is described as a sequence of steps, in each of which a text character is compared to the aligned pattern character. In the case of a mismatch or if an occurrence of the pattern has been verified, the window is shifted along to the next position at which a pattern occurrence is possible. We represent the state of the algorithm before each step by an *information string* uxv, where $u \in \{0, \mathtt{a}\}^k$, $v \in \{0, \mathtt{a}\}^\ell$, and $x \in \{0, \mathtt{A}, \mathtt{b}\}$, describing (part of) the current knowledge the algorithm has about the text characters in the window. A "0" in the information string indicates that no information is available on the corresponding position. An "a" (or a "b") indicates that the character in that position is known to be an a (or a b). An "A" (or a "B") indicates that the character in the corresponding position is known *not* to be an a (or a b). The state can be written in the specified form because, after any necessary window shift, the information string must be consistent with the pattern. Our algorithm makes only "a?" and "b?" queries, and we *choose to forget* any negative information represented by "B". We shall call the $(k + 1)$-st position in the window the b-*position* and all the others a-*positions*. An a-position is always queried for an a. A b-position is always queried for a b.

In terms of the information strings, the algorithm is simply described.

```
IF there is some 0 in the information string
    THEN query the rightmost 0
    ELSE {x = A} query the b-position .
```

This procedure is repeated until the text string is exhausted. To prove the upper bound we first establish the following pair of invariants.

Invariants. (i) If $x = \mathtt{b}$ then $v = \mathtt{a}^\ell$.

(ii) If $x = 0$ then u does not contain the subword $\mathtt{a}^{\ell+1}$.

Invariant (i) holds because x can only become b after an information string of the form $u0\mathtt{a}^\ell$, and following any shift we again have $x \neq \mathtt{b}$.

For Invariant (ii), while $x = 0$ no tests in u are made, and the only a's shifted into u come from v. These are separated from the previous contents of u by the "x" of the previous information string.

Nearly all comparisons can be associated with text positions in the following way. Any query made at an a-position is associated with the corresponding text position. When $x = 0$ and the b-position is queried, a b result is associated with that text position. If the result is B then care is needed since, if $\ell < k$, this result will be represented as a 0 in u. However, in this case a shift of size $\ell + 1$ will be made and, by Invariant (ii), at least one 0 will be shifted out from the information string. The query is associated with the text character corresponding to any one such 0. The remaining case is when $x = \mathtt{A}$ and the b-position is queried. Such a query is not associated with any text position. We note that after such a query a shift of $1 + \max\{k, \ell\}$ is always made, and that the resulting information string will have $x = 0$. Since a window shift is made in any step which changes $x = 0$ to $x = \mathtt{A}$, clearly a cumulative shift of at least $2 + \max\{k, \ell\}$ positions must occur between any two such "extra" queries. The upper bound follows. □

As a corollary we get that Theorem 5.1 is also tight for all members of the $\mathtt{a}^k\mathtt{ba}^\ell$ family to which it can be applied.

COROLLARY 7.2. *For $k, \ell \geq 2$ we have* $c(\mathtt{a}^k\mathtt{ba}^\ell) = c_0^*(\mathtt{a}^k\mathtt{ba}^\ell) = 1 + \frac{1}{\max\{k,\ell\}+2}$.

8. Look-ahead is useful. In this section we present a string matching algorithm, specifically tailored for the string abaa. The algorithm uses a window of size eight and its performance matches the general lower bound obtained for abaa using Theorem 4.1. Thus, $\frac{8}{7} = c(\mathtt{abaa}) = c_4(\mathtt{abaa}) < c_0(\mathtt{abaa}) = \frac{5}{4}$ and abaa is therefore a string for which look-ahead is useful.

The abaa algorithm presented here sheds some light on the intricacy of optimal string matching algorithms. The description of it is quite complicated. Optimal algorithms for longer strings may have even more complicated descriptions.

Algorithm for the string abaa**.** The algorithm requires a window of size 8. A state of the algorithm is given as an information string $\sigma \in \cup_{k=1}^{8}\{0, \mathtt{a}, \mathtt{A}, \mathtt{b}, \mathtt{B}\}^k$, where σ represents information known about the text symbols in (a prefix of) the window. To describe the algorithm, we specify for each state the next query to be made, the amount of shift, and the next state corresponding to the two possible answers. We represent a? queries and b? queries by a single or double underline, respectively, under the appropriate symbol of the information string. For example, in state P in Table 1, an a? query is made at the fourth position in the window.

TABLE 1
The main transition table of the abaa *algorithm.*

state	inf. & query	transitions			
		match		mismatch	
		state	shift	state	shift
P	a00$\underline{0}$	R	0	Q	2
Q	$\underline{0}$A	T	0	U	2
R	a00a00$\underline{0}$	S	0	$Q \oplus F$	5
S	a00a00a$\underline{0}$	$P \oplus H$	7	$T \oplus G$	6
T	aA0$\underline{0}$	$P \oplus D$	3	Q	2
U	$\underline{0}$	P	0	U	1

For certain information strings, the task of finding all pattern occurrences decomposes into two logically disjoint tasks, checking occurrences in some finite prefix and checking in the remainder. Such a state is represented in the tables by a pair of states with the connective $\oplus$. For example, in state R of Table 1, if the seventh symbol is found not to be an a then it is sufficient to check separately for occurrences within the first five positions (state F) and in the text string beginning at the sixth position (state Q).

Table 2 shows, for each finite subproblem which arises in this way, the next query to be made and the number of queries required in the worst case to resolve that subproblem. The latter is computed recursively by following the transitions in Table 2. A "$\surd$" in Table 2 indicates that a full occurrence of the pattern has been found and the treatment of the current subproblem is finished. A "—" indicates that the treatment of the current subproblem has ended without finding such an occurrence.

TABLE 2
The secondary transition table of the abaa *algorithm.*

		new state		
state	inf. & query	match	mismatch	worst-case cost
A	a$\underline{0}$aa	$\surd$	—	1
B	ab$\underline{0}$a	$\surd$	—	1
C	a0a$\underline{0}$	A	—	2
D	a0$\underline{0}$a	A	—	2
E	a$\underline{0}$0aa	A	D	3
F	a$\underline{0}$0a0	C	D	3
G	a00a$\underline{0}$0a	$D \oplus B$	F	4
H	a00a$\underline{0}$0aa	$E \oplus A$	$D \oplus D$	5

In Table 1, the main part of the algorithm is presented. The graph showing the transitions between states of Table 1 is given in Fig. 5. The corresponding number of comparisons to make the transition and finish any consequent subproblem and the resulting shift are shown on each arrow. It can be verified that the worst case corresponds to iterating the cycle PRS, and this proves that $c_4(\texttt{abaa}) \leq \frac{8}{7}$.

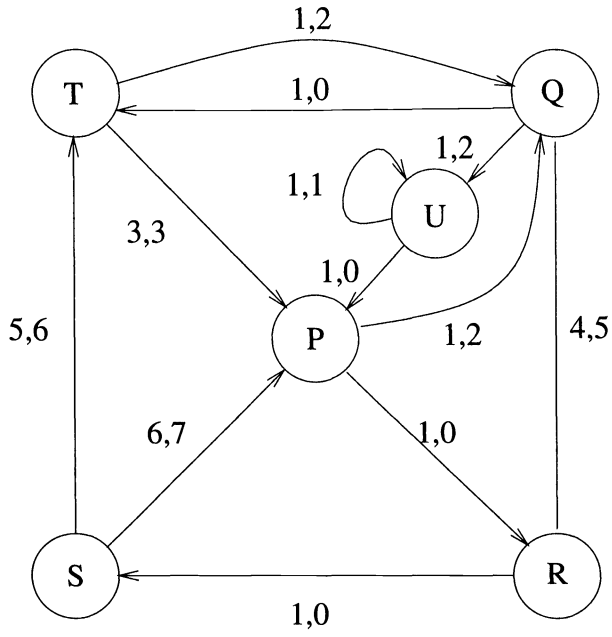


FIG. 5. *The transition diagram of the* abaa *algorithm.*

9. Concluding remarks. What is the hardest string to find? The perhaps disappointing answer is aba (or mum and dad and so on). We know that $c_0^*(\texttt{aba}) = \frac{4}{3}$ and that $c_0^*(w) \leq \frac{4}{3}$

for any string w [GG92]. We believe that $c_0^*(w) < \frac{4}{3}$ whenever $|w| \geq 4$ but this is not established yet (except for $|w| \geq 8$ [CH92]). This would imply that aba and its like are strictly the hardest strings to find. It is interesting to note that while we have shown here that $c^*(\texttt{aba}) = c_0^*(\texttt{aba}) = c_0(\texttt{aba}) = \frac{4}{3}$, the exact value of $c(\texttt{aba})$ is not known yet. We only know that $\frac{6}{5} \leq c(\texttt{aba}) \leq \frac{4}{3}$.

The task of computing the exact value of $c_0^*(w)$ or any of the other three variants, for every given pattern w, seems at present to be very hard. The constants $c_0^*(w)$, $c_0(w)$ can in principle be computed algorithmically since an on-line algorithm can have only a finite number of states representing the current information and only a finite number of possible next queries. Among optimal on-line algorithms for each w, there are some in which the next query depends only on the information state, and there is only a finite though huge number of different algorithms of this kind for every w. The task of finding $c(w)$ and $c^*(w)$ may be even harder. We do not know at present whether $c(w)$ and $c^*(w)$ are always rational, although it would be very odd if they were not.

A small gap still remains between our lower bounds and the upper bounds of Cole and Hariharan [CH92]. While closing this gap will have no practical value, we think that it may reveal many interesting properties of strings and string matching algorithms.

Acknowledgment. The authors thank an anonymous referee for suggestions which improved the paper.

Note added in proof. Shlomit Tassa and the last two authors have recently shown that $c(\texttt{aba}) \leq \frac{5}{4}$, thus proving that text–text comparisons can help.

REFERENCES

[Ah90] A. V. AHO, *Algorithms for finding patterns in strings, Handbook of Theoretical Computer Science, Vol. A*, J. van Leeuwen, ed., Elsevier Science Publishers B.V., New York, 1990, pp. 257–297.

[AG86] A. APOSTOLICO AND R. GIANCARLO, *The Boyer-Moore-Galil string searching strategies revisited*, SIAM J. Comput., 15 (1986), pp. 98–105.

[AC91] A. APOSTOLICO AND M. CROCHEMORE, *Optimal canonization of all substrings of a string*, Inform. and Comput. 95, Academic Press, Orlando, FL, 1991, pp. 76–95.

[BGR90] R. BAEZA-YATES, G.H. GONNET, AND M. REGNIER, *Analysis of Boyer-Moore type string searching algorithms*, Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms, (1990), pp. 328–343.

[BG92] D. BRESLAUER AND Z. GALIL, *Efficient comparison based string matching*, J. Complexity, 9 (1993), pp. 339–365.

[BM77] R. BOYER AND S. MOORE, *A fast string matching algorithm*, Comm. Assoc. Comput. Mach., 20 (1977), pp. 762–772.

[Cole91] R. COLE, *Tight bounds on the complexity of the Boyer–Moore string matching algorithm*, Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms, A. Aggarwal et al., eds., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1991; SIAM J. Comput., 23 (1994), pp. 1075–1091.

[Coll91] L. COLUSSI, *Correctness and efficiency of pattern matching algorithms*, Inform. Comput., 95 (1991), pp. 225–251.

[CH92] R. COLE AND R. HARIHARAN, *Tighter upper bounds on the comparison complexity of string matching*, in preparation. A preliminary version appeared in Proceedings of the 33rd Annual IEEE Symposium on the Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 600–609.

[CHPZ93] R. COLE, R. HARIHARAN, M. PATERSON, AND U. ZWICK, *Which patterns are hard to find?* Proceedings of the 2nd Israeli Symposium on Theory of Computing and Systems, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 59–68.

[CGG90] L. COLUSSI, Z. GALIL, AND R. GIANCARLO, *On the Exact Complexity of String Matching*, Proceedings of the 31st Annual IEEE Symposium on the Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 135–143.

[CCG92] A. CZUMAJ, L. GĄSIENIEC, S. JAROMINEK, W. PLANDOWSKI, AND W. RYTTER, *Speeding Up Two String-Matching Algorithms*, Proceedings of 9th Annual Symposium on Theoretical Aspects of Computer Science, Springer-Verlag, New York, 1992.

[GG91] Z. GALIL AND R. GIANCARLO, *On the Exact Complexity of String Matching: Lower Bounds*, SIAM J. Comput., 20 (1991), pp. 1008–1020.

[GG92] ———, *On the Exact Complexity of String Matching: Upper Bounds*, SIAM J. Comput., 21 (1992), pp. 407–437.

[GO80] L. J. GUIBAS AND A. M. ODLYZKO, *A new proof of the linearity of the Boyer-Moore string searching algorithm*, SIAM J. Comput., 9 (1980), pp. 672–682.

[HS91] A. HUME AND D. SUNDAY, *Fast string searching*, Software – Practice and Experience, 21 (1991), pp. 1221–1248.

[KMP77] D. E. KNUTH, J. MORRIS, AND V. PRATT, *Fast pattern matching in strings*, SIAM J. Comput., 2 (1973), pp. 323–350.

SIAM J. COMPUT.
Vol. 24, No. 1, pp. 46–62, February 1995

003

PLANAR STRONG CONNECTIVITY HELPS IN PARALLEL DEPTH-FIRST SEARCH*

MING-YANG KAO†

Abstract. This paper proves that for a strongly connected planar directed graph of size n, a depth-first search tree rooted at a specified vertex can be computed in $O(\log^5 n)$ time with $n/\log n$ processors. Previously, for planar directed graphs that may not be strongly connected, the best depth-first search algorithm runs in $O(\log^{10} n)$ time with n processors. Both algorithms run on a parallel random access machine that allows concurrent reads and concurrent writes in its shared memory, and in case of a write conflict, permits an arbitrary processor to succeed.

Key words. linear-processor NC algorithms, graph separators, depth-first search, planar directed graphs, strong connectivity, bubble graphs, s-t graphs

AMS subject classifications. 68Q10, 05C99

1. Introduction.

Depth-first search is one of the most useful tools in graph theory [4], [34]. The *depth-first search problem* is the following: Given a graph and a distinguished vertex, construct a tree that corresponds to performing depth-first search in the graph starting from the given vertex.

The parallelization of depth-first search has been studied by numerous authors. Reif showed that lexicographic depth-first search is P-complete even for general undirected graphs [30]. For unordered depth-first search, Smith gave the first NC algorithm for planar undirected graphs [32]. The processor complexity of his algorithm was reduced to linear by He and Yesha [15]. Independently, Ja'Ja and Kosaraju [16] and Shannon [31] also achieved the same result. Aggarwal and Anderson gave a randomized NC algorithm for general undirected graphs [2]. The author studied the problem for directed graphs, and found an NC algorithm with n^4 processors for a planar directed graph of size n [17]. This was followed by the randomized NC algorithm of Aggarwal, Anderson, and Kao for general directed graphs [3].

Recently, Kao and Klein gave an algorithm that computes depth-first search trees in $O(\log^{10} n)$ time with n processors for planar directed graphs that may not be strongly connected [19].

This paper shows that for a strongly connected planar directed graph of size n, a depth-first search tree rooted a specified vertex can be computed in $O(\log^5 n)$ time with $n/\log n$ processors. This algorithm runs on a parallel random access machine that allows concurrent reads and concurrent writes in its shared memory, and in case of a write conflict, permits an arbitrary processor to succeed.

Both the algorithm of this paper and that of the author and Klein use directed graph separators defined by the author [17], and follow the framework of the randomized NC algorithm for general directed graphs [3]. The algorithm in this paper achieves a superior complexity by exploiting topological properties of strongly connected planar directed graphs. The strongly connected components of key subgraphs created in the course of the algorithm have very regular structures. A major task of the algorithm is to recursively maintain and utilize these structures.

This paper is organized as follows. Section 2 reviews basic definitions and relevant facts about planar directed graphs. Section 3 reports new results on computing strongly connected

*Received by the editors March 4, 1992; accepted for publication (in revised form) September 17, 1993. A preliminary version of this paper appeared in the Proceedings of the 1992 International Computer Symposium, Taichung, Taiwain, Republic of China, December 13–15, 1992, pp. 309–316.

†Department of Computer Science, Duke University, Durham, North Carolina 27708 (kao@cs.duke.edu).
This research is supported in part by National Science Foundation Grants CCR-8909323 and CCR-9101385.

components. Section 4 discusses directed graph separators; a more detailed discussion about separators is given in the Appendix. Section 5 details this paper's depth-first search algorithm for strongly connected planar directed graphs.

2. Basics of planar directed graphs. A *planar* directed graph is one that can be embedded on a plane such that the edges intersect only at common end vertices or start vertices [6], [7], [14], [35]. A *plane digraph* is a planar directed graph with a given planar embedding.

For subtle technical reasons, a planar graph in this paper may have multiple edges but does not have loop edges.

2.1. Strong graphs. For brevity, a strongly connected component of a directed graph is simply called a *strong component.* Similarly, a strongly connected plane digraph with at least one vertex is simply called a *strong graph.*

The goal of this paper is to efficiently compute a depth-first search spanning tree of a strong graph rooted at a specified vertex.

2.2. Faces, boundaries, and orientations. Let G be a connected plane digraph. If the vertices and edges of G are deleted from its embedding plane, then the plane is divided into disconnected regions. Exactly one of the regions is infinite; all others are finite. Each region is called a *face* of G. The infinite region is the *external* face; the finite regions are the *internal* faces.

Let f be a face of G. The *boundary* of f, denoted by $\mathcal{B}(f)$, is the set of edges and vertices surrounding f. If G contains at least two vertices, by its connectivity, $\mathcal{B}(f)$ can be arranged into a unique undirected cycle by having an observer stay inside f and walk around $\mathcal{B}(f)$ once. This cycle is called the *boundary* cycle of f. It may not be edge-simple.

Let e be a boundary edge of f. The *orientation* of e with respect to f is defined as follows:

- Case (1): f is the external face. The edge e is *positive* (respectively, *negative*) with respect to f if it points in the counterclockwise (respectively, clockwise) direction on the boundary cycle of f.
- Case (2): f is an internal face. The edge e is *positive* (respectively, *negative*) with respect to f if it points in the clockwise (respectively, counterclockwise) direction on the boundary cycle of f.

2.3. Holes, boundaries, and orientations. Let G be a connected plane digraph. Let H be a connected subgraph of G. If the vertices and edges of H are removed from the embedding plane of G, then the plane is divided into disconnected regions. Exactly one of the regions is infinite; the others are all finite. Each region is called a *hole* of H. The infinite region is the *external* hole; the finite regions are the *internal* holes.

Let X be a hole of H. The *boundary* of X, denoted by $\mathcal{B}(X)$, is the set of vertices and edges surrounding X. If H contains at least two vertices, by its connectivity, $\mathcal{B}(X)$ can be arranged into a unique undirected cycle by having an observer stay inside X and walk around $\mathcal{B}(X)$ exactly once. This cycle is called the *boundary* cycle of X. It may not be edge-simple.

Let e be a boundary edge of X. The *orientation* of e with respect to X is defined as follows:

- Case (1): X is the external hole of H. The edge e is *positive* (respectively, *negative*) with respect to X if it points in the counterclockwise (respectively, clockwise) direction on the boundary cycle of X.
- Case (2): X is an internal hole of H. The edge e is *positive* (respectively, *negative*) with respect to X if it points in the clockwise (respectively, counterclockwise) direction on the boundary cycle of X.

2.4. Combinatorial embeddings and data structures. Let G be a connected planar directed graph. Algorithmically, a planar embedding of G is encoded by the boundary of its external face and the clockwise cyclic order of the edges incident with each vertex. Such an encoding is called a *combinatorial* planar embedding of G. Topologically, a planar embedding is uniquely specified by its corresponding combinatorial embedding.

The cyclic edge incidence in a combinatorial embedding is further encoded by the following data structure: For each vertex, there is a doubly linked circular list consisting of the edges incident with that vertex in the clockwise order. These lists can be used to efficiently trace the boundary cycles of the faces of G. They can also be used to trace the boundary cycles of the holes of a connected subgraph.

Given a connected planar directed graph of size n, a combinatorial planar embedding can be computed in $O(\log n)$ time with $n \log\log n / \log n$ processors on a deterministic Arbitrary-CRCW PRAM [29].

2.5. Planar embeddings induced by vertex contraction. In this paper, vertex contraction contracts only connected vertex subsets of a connected plane digraph. This ensures that planarity is preserved.

For technical reasons, all multiple edges created by vertex contraction are kept while all loop edges are deleted.

Let G be a connected plane digraph. Let H be the subgraph induced by a connected vertex subset of G. Let G' be the graph constructed from G by contracting H into a vertex H'. If H consists of at most one vertex, then G' and G are the same. Otherwise a planar embedding for G' is specified as follows:

- For every vertex $u \notin H$, the clockwise cyclic order of the edges incident with u is the same in G and G'.
- The edges around each nonempty hole X of H stay together around H', and their clockwise cyclic order around H' is the same as their cyclic order around the boundary cycle of X in the negative direction of X.
- All uncontracted edges on the boundary of the external face of G remain on that of G', and have the same orientations with respect to both external faces.
- If H contains a boundary vertex of the external face of G, then H' is on the boundary of the external face of G'.

In general such a planar embedding is not unique. Any planar embedding that fits this construction is suitable for the purposes of this paper.

LEMMA 2.1. *Given a connected plane digraph of size n, a planar embedding induced by contracting a disjoint family of connected vertex subsets can be computed in* $O(\log n)$ *time with* $n/\log n$ *processors.*

Remark. If vertex contraction is required to delete all multiple edges that it creates, then computing an induced embedding may take more than linear space to achieve $O(\log n)$ time on $n/\log n$ processors.

Proof. The edges around a new vertex of G' are collected by processing the doubly linked circular lists of the combinatorial embedding of G. The computation takes $O(\log n)$ time and $n/\log n$ processors, using optimal parallel algorithms for list ranking [5], [9], [13], prefix computation [23], [24], tree contraction [1], [8], [11], [10], [22], [27], [28], and planar connectivity [12]. □

3. New results for computing strong components. This section first reviews previous results on computing strong components and directed spanning trees, and then reports new results on how to compute the strong components of a graph that is obtained by deleting or contracting a subgraph.

3.1. Directed spanning trees.

THEOREM 3.1 [20]. *For a strong graph of size n, a directed spanning tree rooted at a specified vertex can be computed in $O(\log^2 n)$ time with $n/\log n$ processors on a deterministic* Arbitrary-CRCW PRAM.

3.2. Compactness and strong components. Let H_1 and H_2 be nonempty connected subgraphs of a connected plane digraph. H_1 *encloses* H_2 if H_2 is in an internal hole of H_1.

A *compact* strong component of a connected plane digraph is one that encloses no strong component. For a plane digraph that may not be connected, a strong component is *compact* if it is compact in the connected component that contains it.

THEOREM 3.2 [18]. *For a plane digraph of size n with α noncompact strong components, the strong components can be computed in $O(\lceil \log_2(\alpha+2) \rceil \cdot \log^2 n)$ time with $n/\log n$ processors on a deterministic* Arbitrary-CRCW PRAM.

3.3. Deleting a vertex subset from a strong graph. Let G be a directed graph. Let H be a subgraph or vertex subset of G. Let $G - H$ denote the graph obtained by removing from G the vertices in H and their incident edges.

THEOREM 3.3. *Assume that G is a strong graph of size n and H is a subgraph with γ connected components. Given $G - H$, the strong components of $G - H$ can be computed in $O(\lceil \log_2(\gamma+2) \rceil \cdot \log^2 n)$ time with $n/\log n$ processors.*

Proof. The noncompact strong components of $G - H$ can be counted using the three facts below. Let W be a noncompact strong component of $G - H$.

- By G's planarity and strong connectivity, at least one connected component of H is enclosed by W in G.
- By G's planarity, if some connected component of H is enclosed in G by both W and a distinct strong component W' of $G - H$, then one of W and W' encloses the other in G.
- By G's strong connectivity and planarity, some connected component H' of H is enclosed by W in G but not by any strong component of $G - H$ that is itself enclosed by W in G.

The mapping from W to H' is a one-to-one mapping from the noncompact strong components of $G - H$ to the connected components of H. Thus, $G - H$ has at most γ noncompact strong components, and this theorem follows from Theorem 3.2. □

3.4. Deleting an edge subset from a strong graph. Let G be a strong graph. Let U be a vertex subset of G. Let D be a subset of the edges in G incident with U. Let G' be the graph obtained by deleting from G the edges in D.

Let W be a connected subgraph of G. A vertex of G is *absorbed* by W in G if it either is a vertex in W or is enclosed by W in G.

THEOREM 3.4. *Let n be the size of G. Let ω be the number of vertices in U. Given G', the strong components of G' can be computed in $O(\lceil \log_2(2\omega+2) \rceil \cdot \log^2 n)$ with $n/\log n$ processors.*

Proof. The noncompact strong components of G' can be counted using the three facts below. Let W be a noncompact strong component of G'.

- By G's planarity and strong connectivity, some vertex in U is absorbed by W in G.
- By G's planarity, if some vertex in U is absorbed by both W and another strong component W' of G', then one of W and W' encloses the other in G.
- By G's strong connectivity and planarity, some $u \in U$ is absorbed by W in G but not by any strong component of G' that is itself enclosed by W in G, possibly except a unique component W'' that contains u as a boundary vertex of its external face.

The mapping from W to u maps at most two noncompact strong components, namely W and W'', of G' to u. Thus, G' has at most 2ω noncompact strong components, and this theorem follows from Theorem 3.2. □

3.5. Contracting connected vertex subsets in an acyclic graph. Let G be an acyclic connected plane digraph. Let H be a subgraph of G. Let G' be the graph obtained from G by contracting each connected component of H into a vertex.

THEOREM 3.5. *Assume that G is of size n and H has γ connected components. Given G', the strong components of G' can be computed in $O(\lceil \log_2(\gamma+2) \rceil \cdot \log^2 n)$ time with $n/\log n$ processors.*

Proof. Since a noncompact strong component has at least two vertices, by the acyclicity of G each noncompact strong component of G' contains at least one contracted connected component of H. Because two distinct strong components of G cannot contain the same contracted connected component of H, G' has at most γ noncompact strong components. This theorem then follows from Theorem 3.2. □

4. Directed graph separators. Intuitively, a *separator* of a graph is a subgraph whose removal disconnects the graph into small pieces [25], [26].

4.1. Cycle separators and k-path separators. Most of the works on parallel depth-first search rely on finding some form of graph separator. The algorithms for planar undirected graphs employ undirected cycle separators [15], [16], [32]. The algorithm for general undirected graphs uses path separators [2].

The notion of a directed graph separator was originally introduced for depth-first search in planar directed graphs [17]. It was then used for general directed graphs [3]. Here it is tailored for a strongly connected directed graph G:

- A vertex subset or subgraph is *heavy* (respectively, *light*) for G if it has more than (respectively, at most) two thirds of the vertices in G.
- A *separator* of G is a set S of vertices such that no strong component in $G - S$ is heavy for G.
- A *cycle* separator is a vertex-simple directed cycle whose vertices form a separator. A single vertex is considered a cycle of length zero. Thus, if the removal of a vertex separates a graph, the vertex is a cycle separator.
- For a positive integer k, a *k-path* separator is a set of k vertex-disjoint vertex-simple directed paths whose vertices form a separator. A 1-path separator is simply called a *path* separator.

The author showed that every directed graph has a directed path separator and a directed cycle separator [17]. These results are included in the Appendix. For a graph of size n, the proofs of these results yield a sequential algorithm that computes a path separator in optimal $O(n)$ time, and an algorithm that computes a cycle separator in $O(n \log n)$ time.

Aggarwal, Anderson, and Kao improved to $O(n)$ the sequential time for computing a directed cycle separator [3]. They also showed that computing cycle separators and computing depth-first search trees are NC-equivalent.

4.2. Computing cycle separators for strong graphs.

LEMMA 4.1. *Given a strong graph of size n, a two-path separator can be computed in $O(\log^2 n)$ time with $n/\log n$ processors.*

Proof. Let T be a directed spanning tree of the given graph. By the work of Lipton and Tarjan [25], there exist two vertices x and y such that the two tree paths of T from the root to x and from the root to y form a separator. A two-path separator is easily obtained from these tree paths. By Theorem 3.1, T can be computed in $O(\log n)$ time with $n/\log n$ processors. Then x and y can be found in $O(\log n)$ time with $n/\log n$ processors [21]. □

LEMMA 4.2. *Given a strong graph of size n, a path separator can be computed in* $O(\log^3 n)$ *time with* $n/\log n$ *processors.*

Proof. Let G be the given graph. First, use Lemma 4.1 to obtain a two-path separator for G, and then use the procedure MergeTwoPaths in Fig. 1 to compute a path separator. To combine P and Q, MergeTwoPaths first deletes the maximum subpath from the u_p-end of P while maintaining the separator property of P and Q. It then similarly processes Q so that the joining path R exists. The proof for the correctness of MergeTwoPaths is similar to that of Theorem A.2 in §A.3.

Procedure MergeTwoPaths
Input: a strong graph G, and a two-path separator $P = u_1, \ldots, u_p$ and $Q = v_1, \ldots, v_q$.
Output: a path separator S of G.
begin
1. Let s be the largest index such that some strong component Z_s of $G - (\{u_1, \ldots, u_{s-1}\} \cup Q)$ is heavy for G.
   (*Remark.* $u_s \in Z_s$.)
2. **if** s does not exist **then return** $S = Q$.
3. Let P' be the path $u_1, \ldots, u_s$.
   (*Remark.* P' and Q form a two-path separator.)
4. Let t be the smallest index such that some strong component Z_t of $G - (P' \cup \{v_{t+1}, \ldots, v_q\})$ is heavy for G.
   (*Remark.* $v_t \in Z_t$.)
5. **if** t does not exist **then return** $S = P'$.
6. Let Q' be the path $v_t, \ldots, v_q$.
   (*Remark.* P' and Q' form a two-path separator.)
7. Compute a directed path R in $Z_s \cup Z_t$ from u_s to v_t.
   (*Remark.* $Z_s \cup Z_t$ is strongly connected.)
8. Let S be the directed path formed by P', R, Q'.
   (*Remark.* S is vertex-simple.)
9. **return** S.

end.

FIG. 1. *A procedure for merging a two-path separator into a path separator.*

As for the complexity, it suffices to show that MergeTwoPaths runs in $O(\log^3 n)$ time with $n/\log n$ processors. Z_s, Z_t, s, and t are computed by binary search and Theorem 3.3 in $O(\log^3 n)$ time with $n/\log n$ processors. R is obtained by computing a divergent directed spanning tree rooted at u_s in the subgraph induced by $Z_s \cup Z_t$. This uses Theorem 3.1 and takes $O(\log^2 n)$ time with $n/\log n$ processors. Thus, the complexity of MergeTwoPaths is as stated. □

THEOREM 4.3. *Given a strong graph of size n, a cycle separator can be computed in* $O(\log^3 n)$ *time with* $n/\log n$ *processors.*

Proof. The proof is similar to those of Lemma 4.2 and Theorem A.2. □

5. Parallel depth-first search. Section 5.1 gives an overview of this paper's algorithm for performing depth-first search in a strong graph. Sections 5.2–5.6 discuss key techniques used in the algorithm. Section 5.8 details the algorithm.

5.1. An overview. Let G be a strong graph. Let r be a vertex in G. The goal is to construct a depth-first search spanning tree rooted at r for G. Such a tree will be recursively constructed using cycle separators.

First, compute a cycle separator of G. Then compute a path separator starting from r by finding a directed path from r to the cycle separator. This path and the cycle separator form a path separator S after an appropriate edge on the cycle separator is removed. S will be a branch of the final depth-first search tree.

Let $G' = G - S$, i.e., the remaining graph that is not searched by S. Suppose that the search is continued in G' starting from a vertex r' that is the end vertex of an edge pointing from the last vertex of S. This time the search recurses on the subgraph $B_{r'}$ that consists of all the vertices reachable from r' via directed paths in G'. The graph $B_{r'}$ is called a *dangling* subgraph. (See §5.5.)

Because S is a separator of G, the strong components of G' are all light for G. However, $B_{r'}$ may contain several such strong components. Consequently, $B_{r'}$ may still be too large for small depth recursion. To avoid this problem, a set of directed paths is removed from G such that the remaining directed graph has small dangling subgraphs. These removed paths will form a subtree, called a *partial* depth-first search tree, in the final depth-first search tree. (See §5.5.)

A dangling subgraph of a strong graph is a special kind of a graph called a *bubble* graph. The structures of a bubble graph can be exploited to efficiently process the dangling subgraphs. (See §5.2.)

A strong graph is in fact a special case of a bubble graph. The depth-first search algorithm in this paper actually takes a bubble graph as input, and computes a depth-first search tree by recursing on bubble subgraphs. (See §5.7.)

5.2. Bubble graphs. A strong component of a directed graph is a *sink* component if it has no outgoing edges to any other strong component.

Let B be a plane digraph. B is called a *bubble* graph rooted at a vertex r if the following conditions hold:

- Every vertex in B can be reached from r via directed paths.
- The vertex r is a boundary vertex of the external face of B, and every sink component of B contains at least one boundary vertex of the external face.

For example, since a strong graph is the only strong component of itself, it is trivially a bubble graph rooted at any boundary vertex of its external face.

From this point onwards, it is assumed that a bubble graph has a specified root. For brevity, that root will not be explicitly mentioned unless there is a risk of ambiguity.

LEMMA 5.1. *Let B be a bubble graph rooted at r. Let n be the size of B.*

1. *The strong components of B can be computed in $O(\log^2 n)$ time with $n/\log n$ processors.*
2. *A directed spanning tree of B rooted at r can be computed in $O(\log^2 n)$ time with $n/\log n$ processors.*

Proof. Statement 1 follows from Theorem 3.2 and the fact that a bubble graph has no noncompact strong components. To prove Statement 2, let B' be the strong graph obtained from B by adding a directed edge from each sink component of B to r via the external face of B. Note that a directed spanning tree of B' rooted at r is also one for B. The graph B' and a desired tree in B' can be computed by means of Statement 1 and Theorem 3.2, respectively, both within the desired complexity. □

5.3. Heavy bubble graphs and splitting components. For a directed graph G and a vertex subset (a vertex or a subgraph) H of G, let $\mathcal{R}(H, G)$ denote the set of vertices that H

can reach via directed paths in G. For all strong components W_1 and W_2 of G, W_2 is called a *descendant component* of W_1 if $W_1 \neq W_2$ and W_1 can reach W_2 via directed paths.

Let k be a positive integer. A bubble graph is called *k-heavy* (respectively, *k-light*) if it has more than (respectively, at most) k vertices. A strong component W of a k-heavy bubble graph B is a *k-splitting* component of B if $|\mathcal{R}(W, B)| > k$ and no descendant component of W satisfies this inequality.

The depth-first search algorithm of this paper will find a cycle separator of a k-splitting component of B, and then use the separator to break B into bubble subgraphs with smaller k-splitting components.

5.4. Computing a splitting component via *s-t* graphs. An acyclic plane digraph is called an *s-t* graph if it has a unique source and a unique sink, and they are on the boundary of its external face.

THEOREM 5.2. *Given a k-heavy bubble graph of size n, a k-splitting component can be computed in $O(\log^2 n)$ time with $n/\log n$ processors.*

Proof. Let B be a k-heavy bubble graph. A k-splitting component of B is computed by converting B into an *s-t* graph as follows.

- Let B_1 be the graph obtained by adding a vertex t in the external face of B and a directed edge pointing to t from each sink component of B.
- Let B_2 be the graph obtained from B_1 by contracting each strong component of B into a vertex. The vertices in B_2 are assigned weights. The weight of t is 0. For each vertex w that is contracted from a strong component W of B, the weight of w is the number of vertices in W.

Note that B_2 is an *s-t* graph. Its sink is t, and its source is contracted from the strong component of B that contains B's specified root.

By Lemmas 5.1(1) and 2.1, B_2 can be computed in $O(\log^2 n)$ time with $n/\log n$ processors. Next, the cardinality of $\mathcal{R}(W, B)$ for each strong component W of B is computed in $O(\log^2 n)$ time on $n/\log n$ processors by applying to B_2 a slight generalization of the descendant counting algorithm of Tamassia and Vitter for *s-t* graphs [33]. With the cardinality of $\mathcal{R}(W, B)$ computed, a k-splitting component of B is identified in a straightforward manner in $O(\log n)$ time with $n/\log n$ processors. □

5.5. Partial depth-first search trees and dangling subgraphs. Let B be a bubble graph. A *partial* depth-first search tree of B is a subtree of a depth-first search tree of B such that both trees are rooted at the specified root of B.

Let T be a partial depth-first search tree of B. Let $x_1, x_2, \ldots, x_t$ be the vertices of T listed in the postorder traversal sequence of depth-first search, i.e., in this sequence x_i is marked right after all its descendants in T are marked.

For each x_i, let $y_{i,1}, \ldots, y_{i,k_i}$ be the vertices that are not in T but are the end vertices of the edges pointing from x_i. The order of $y_{i,1}, \ldots, y_{i,k_i}$ is arbitrary. This is the postorder that will be used to search $B - T$ starting from x_i. A y vertex may have several different indices if it is adjacent from several x vertices.

The *dangling* subgraph of B, denoted by $\mathcal{D}(i, j)$, with respect to (i, j) and T is the subgraph induced by the vertices in $B - T$ that can be reached from $y_{i,j}$ but not from any y vertex before $y_{i,j}$ in the intended postorder traversal sequence of the y vertices, i.e., $\mathcal{D}(i, j) = \mathcal{R}(y_{i,j}, B - T) - \cup\{\mathcal{R}(y_{i',j'}, B - T) | i' < i \vee (i' = i \wedge j' < j)\}$.

For each nonempty $\mathcal{D}(i, j)$, the directed edge from x_i to $y_{i,j}$ is called the *dangling* edge associated with $\mathcal{D}(i, j)$.

The next two lemmas provide a natural way of extending T into a complete depth-first search tree by recursing on the nonempty dangling subgraphs in parallel.

LEMMA 5.3. *Let r be the specified root of B. Let H be a connected subgraph of B that contains r. Let u be a vertex that is not in H but is adjacent to or from H. Then the subgraph of B induced by $\mathcal{R}(u, B - H)$ is a bubble graph rooted at u.*

Proof. Let B_u be the subgraph of B induced by $\mathcal{R}(u, B - H)$. First, by the definition of B_u, the vertex u can reach every vertex in B_u via directed paths in B_u. Next, because r is a boundary vertex of the external face of B, by the connectivity of H and by the relationship of r, H, and u, the vertex u is a boundary vertex of the external face of B_u.

Let W be a sink component of B_u. The following discussion shows that W contains a boundary vertex of the external face of B_u. Because B is a bubble graph, B contains a directed path $Q = y_1, \ldots, y_s$ from W to a boundary vertex on the external face of B. There are two cases based on whether Q intersects H or not.

- Case (1): Q does not intersect H. Then Q lies in B_u. Because W is a sink component of B_u, the path Q lies in W. Therefore, W contains the last vertex of Q, which is a boundary vertex on the external face of B_u.
- Case (2): Q intersects H. Let y_s be the first vertex of Q that is in H. Because $r \in H$ is a boundary vertex on the external face of B, by the adjacency of H and y_{s-1}, the vertex y_{s-1} is a boundary vertex of the external face of B_u. Furthermore, because W is a sink component of B_u, the vertex y_{s-1} is in W. □

LEMMA 5.4. *Let Ω be the set of dangling subgraphs of B with respect to T. Let L be the set of the associated dangling edges. Then the following statements are true*:

1. *Each nonempty $\mathcal{D}(i, j)$ is a bubble graph rooted at $y_{i,j}$. Hence $y_{i,j}$ is chosen to be the specified root of $\mathcal{D}(i, j)$.*
2. *The dangling subgraphs are disjoint.*
3. *A depth-first search tree of B can be formed by T, L, and a depth-first search tree for each dangling subgraph with at least two vertices.*

Proof. The first statement is obtained by recursively applying Lemma 5.3. The other two statements are straightforward. □

5.6. Computing dangling subgraphs with respect to a path. This section shows how to compute the dangling subgraphs with respect to a partial depth-first search tree that is a path. The computation is based on two bisection strategies using the subroutines in Figs. 2 and 3.

5.6.1. Analyzing the subroutine in Fig. 2.

LEMMA 5.5. *Let Y be the set of the end vertices of the edges in B that point from P_1 to $B - P$. Then, $\mathcal{R}(Y, B - P) = W - \{w\}$.*

Proof. The proof has two directions.

$\mathcal{R}(Y, B - P) \subseteq W - \{w\}$: Let $y \in \mathcal{R}(Y, B - P)$. Then, $B - P_2$ contains a directed path Q_1 from P_1 to y. Note that Q_1 contains no edge from D. On the other hand, because A_1 is strongly connected, there is a directed path Q_2 from y to P such that Q_2 and P intersect at only one vertex. Because Q_2 contains no outgoing edges from P, it has no edge from D. Let Q be the directed path formed by Q_1 and Q_2. Because Q goes from P to P, it becomes a directed cycle in A_2 that contains w. Because Q contains no edge from D. it remains a directed cycle in A_3. Therefore, $y \in W - \{w\}$.

$\mathcal{R}(Y, B - P) \supseteq W - \{w\}$: Let $z \in W - \{w\}$. Then A_3 contains a vertex-simple directed path R_1 from w to z. Let R_2 be a directed path in A_1 that corresponds to R_1 and intersects P only at one vertex x. Note that x is the start vertex of R_1. Because A_1 is constructed by adding incoming edges to r and because the edges in D are removed from A_3, the vertex x is in P_1 and the edges of R_2 are in $B - P_2$. Therefore, $z \in \mathcal{R}(Y, B - P)$. □

LEMMA 5.6. *Let B be a bubble graph of size n. Then the procedure in Fig. 2 correctly computes an output as specified in $O(\log^2 n)$ time with $n/\log n$ processors.*

Procedure SubOneComputeDSG
Input: a bubble graph B rooted at r, and a vertex-simple directed path $P = x_p, \ldots, x_1$ with $x_p = r$ and $p \geq 2$.
Output: two graphs B_1 and B_2 constructed from B, and two paths $P_2 = x_p, \ldots, x_{\lceil p/2 \rceil + 1}$ and $P_1 = x_{\lceil p/2 \rceil}, \ldots, x_1$ with following properties:

1. B_1 and B_2 are bubble graphs rooted at $x_{\lceil p/2 \rceil}$ and x_p, respectively.
2. P_1 and P_2 are vertex-simple directed paths, respectively, in B_1 and B_2 starting from their specified roots.
3. The nonempty dangling subgraphs and the associated dangling edges of B with respect to P are exactly those of B_1 with respect to P_1 and those of B_2 with respect to P_2.
4. The total size of B_1 and B_2 is at most the size of B.

begin

1. Let $q = \lceil p/2 \rceil$.
2. Let P_1 be the subpath of P formed by $x_q, \ldots, x_1$.
3. Let P_2 be the subpath of P formed by $x_p, \ldots, x_{q+1}$.
4. Let D be the set of edges in B that point from P_2 to $B - P$.
5. **if** B is strongly connected
   then let $A_1 = B$
   else let A_1 be the graph obtained from B by adding a directed edge from each sink component to r via the external face of B.
6. Let A_2 be the graph obtained from A_1 by contracting P into a vertex w. (*Remark.* The edges in D are now outgoing edges of w.)
7. Let A_3 be the graph obtained from A_2 by deleting the edges in D.
8. Let W be the strong component in A_3 that contains w.
9. Let B_1 be the subgraph of B induced by $(W - \{w\}) \cup P_1$.
10. Let B_2 be the graph obtained from B by contracting $(W - \{w\}) \cup P_1 \cup \{x_{q+1}\}$ into x_{q+1}.
11. **return** B_1, B_2, P_1, and P_2.

end.

FIG. 2. *The first subroutine for computing dangling subgraphs.*

Proof. The second and the fourth output property of the procedure are straightforward. The other two properties are shown below.

- Property 1: By Lemma 5.5, B_1 is the subgraph of B induced by $\mathcal{R}(x_q, B - P_2)$. Therefore, by Lemma 5.3, B_1 is a bubble graph rooted at x_q. Then, because B_2 is obtained from a bubble graph by contracting a connected vertex subset, it is a bubble graph rooted at x_p.
- Property 3: By Lemma 5.5, the nonempty dangling subgraphs and the associated dangling edges of B with respect to P are exactly those of B_1 with respect to P_1 and those of $B - B_1$ with respect to P_2. Because $B - B_1$ may or may not be a bubble graph, B_1 is contracted instead of deleted. The contraction of B_1 into x_{q+1} may add new edges in B_2 from $B - (B_1 \cup P_2)$ to x_{q+1}. However, this contraction creates no new edges in B_2 from x_{q+1} to $B - (B_1 \cup P_2)$. Such edges would point from B_1

Procedure SubTwoComputeDSG
Input: a bubble graph B rooted at r with $k \geq 2$ edges outgoing from r.
Output: two graphs B_1 and B_2 constructed from B with the following properties:

1. B_1 and B_2 are bubble graphs rooted at r.
2. The nonempty dangling subgraphs and the associated dangling edges of B with respect to r are exactly those of B_1 and B_2 with respect to r, where r is considered as a path of a single vertex.
3. The outdegrees of r in B_1 and B_2 are at most $\lceil k/2 \rceil$ and $\lfloor k/2 \rfloor$, respectively.
4. The total size of B_1 and B_2 is at most the size of B.

begin

1. Let $h = \lceil k/2 \rceil$.
2. Let $y_1, \ldots, y_k$ be the k end vertices of the outgoing edges of r in B.
3. Let D be the set of the edges from r to $y_{h+1}, \ldots, y_k$ in B.
4. **if** B is strongly connected
   then let $A_1 = B$
   else let A_1 be the graph obtained from B by adding a directed edge from each sink component to r via the external face of B.
5. Let A_2 be the graph obtained from A_1 by contracting $\{r, y_1, \ldots, y_h\}$ into a vertex w.
   (*Remark.* The edges in D are now outgoing edges of w.)
6. Let A_3 be the graph obtained from A_2 by deleting the edges in D.
7. Let W be the strong component in A_3 that contains w.
8. Let B_1 be the subgraph of B induced by $(W - \{w\}) \cup \{r, y_1, \ldots, y_h\}$ without the edges in D.
9. Let B_2 be the graph obtained from B by contracting $(W - \{w\}) \cup \{r, y_1, \ldots, y_h\}$ into r.
10. **return** B_1 and B_2.

end.

FIG. 3. *The second subroutine for computing dangling subgraphs.*

and thus their end vertices would have been included in B_1. Therefore, contracting B_1 into x_{q+1} does not change the nonempty dangling subgraphs and the associated dangling edges of $B - B_1$ with respect to P_2.

As for the complexity, Steps 1–4 can be done in $O(\log n)$ time with $n/\log n$ processors. Step 5 can be done via Lemma 5.1(1) in $O(\log^2 n)$ time with $n/\log n$ processors. Step 6 is done via Lemma 2.1 in $O(\log n)$ time with $n/\log n$ processors. Step 7 can be done in $O(\log n)$ time with $n/\log n$ processors. Because A_1 is a strong graph, A_2 remains a strong graph. Because the edges of D are adjacent to w in A_2, by Theorem 3.4, W can be computed in $O(\log^2 n)$ time with $n/\log n$ processors. Step 9 can be done in $O(\log n)$ time with $n/\log n$ processors. Step 10 can be done via Lemma 2.1 in $O(\log n)$ time with $n/\log n$ processors. Thus, the total complexity of the procedure in Fig. 2 is as stated. □

5.6.2. Analyzing the subroutine in Fig. 3.

LEMMA 5.7. *Let B be a bubble graph of size n. Then the procedure in Fig. 3 correctly computes an output as specified in $O(\log^2 n)$ time with $n/\log n$ processors.*

Proof. The proof is similar to those of Lemmas 5.6 and 5.3. A subtle point is as follows. B_1 may contain some of $y_{h+1}, \ldots, y_k$. The dangling subgraphs of B with respect to such vertices are empty. Therefore, the edges in D can be deleted from B_1 without affecting its nonempty dangling subgraphs. This deletion ensures that the outdegree of r in B_1 is $\lceil k/2 \rceil$.

Remark. The procedure in Fig. 3 can be simplified by replacing Steps 5–9 with the following steps. This simplification decreases the apparent symmetry between the procedures in Figs. 2 and 3.

6′. Let A_3' be the graph obtained from A_1 by deleting the edges in D.
7′. Let W' be the strong component in A_3' that contains r.
8′. Let B_1 be the subgraph of B induced by W' without the edges in D.
9′. Let B_2 be the graph obtained from B by contracting B_1 into r. □

5.6.3. Computing dangling subgraphs. The next theorem uses the procedures in Figs. 2 and 3 to compute the nonempty dangling subgraphs with respect to a path.

THEOREM 5.8. *Let B be a bubble graph of size n. Let P be a vertex-simple directed path of B starting from its specified root. Then the nonempty dangling subgraphs and the associated dangling edges of B with respect to P can be computed in $O(\log^3 n)$ time with $n/\log n$ processors.*

Proof. The computation is divided into two phases:

- Phase 1 iteratively applies SubOneComputeDSG to B and P to bisect P. In $O(\log n)$ iterations, a collection of bubble subgraphs of B is obtained such that each subgraph B' contains only one vertex of P.
- Phase 2 iteratively applies SubTwoComputeDSG to each B' to bisect the outdegree of its root. In $O(\log n)$ iterations, a collection of even smaller bubble subgraphs of B is obtained. Each subgraph B'' is rooted at a vertex y. B'' either has exactly one outgoing edge from y or consists of only y. If B'' consists of only y, then its corresponding dangling subgraph of B is empty. Otherwise, $B'' - \{y\}$ is a nonempty dangling subgraph of B. Its associated dangling edge is the outgoing edge of y in B''.

The correctness and complexity of this computation follow directly from Lemmas 5.6 and 5.7. □

5.7. Parallel depth-first search in bubble graphs. Figure 6 details this paper's algorithm for performing depth-first search in a bubble graph. Its subroutines are described in Figs. 5 and 4.

LEMMA 5.9. *Assume that B is a bubble graph of size n. The procedure in Fig. 4 correctly computes an output as specified in $O(\log^3 n)$ time with $n/\log n$ processors.*

Proof. Let S' be the set of vertices in $S \cap W$. Assume that some dangling subgraph B' of B with respect to S is $2m/3$-heavy. Because B has at most m vertices, B' is the only $2m/3$-heavy dangling subgraph of B. Note that W is the only $2m/3$-splitting component of B included in the subgraph $\mathcal{R}(W, B)$. Thus, the $2m/3$-splitting components of B' must be strong components of $W - S'$. Then, because S' is a separator of W, the output property of the procedure holds.

As for the complexity of the procedure, Steps 1, 5, and 6 are obvious. Step 2 employs Theorem 5.2. Step 3 uses Theorem 4.3. Step 4 is done by means of Lemma 5.1(2). Steps 7 and 8 use Theorem 5.8. Thus, the total complexity is as stated. □

LEMMA 5.10. *Let B be a bubble graph of size n. The procedure in Fig. 5 correctly computes an output as specified in $O(\log^4 n)$ time with $n/\log n$ processors.*

Proof. At Step 2 in Fig. 5, by the output properties of SplitHeavyDSG, the $2m/3$-splitting components of the input bubble graph to each call of SplitHeavyDSG are all light subgraphs of a $2m/3$-splitting component of the input bubble graph to the previous call. Thus, the while loop at Step 2 iterates $O(\log n)$ time, and this lemma follows directly from Lemma 5.9. □

Procedure SplitHeavyDSG
Input: a positive integer m, and a $2m/3$-heavy bubble graph B that has at most m vertices.
Output: a partial depth-first search tree S of B, the set Δ of all dangling subgraphs of B with respect to S, and the set K of the associated dangling edges, where S satisfies either property below:

1. No subgraph in Δ is $2m/3$-heavy.
2. Some subgraph in Δ is $2m/3$-heavy and its $2m/3$-splitting components are all light subgraphs of a single $2m/3$-splitting component W of B.

begin

1. Let r be the specified root of B.
2. Let W be a $2m/3$-splitting component of B.
3. Let C be a directed cycle separator of W.
4. Let P be a vertex-simple directed path in B from r to C such that P and C intersect at only one vertex u.
5. Let e be the edge on C pointing to u.
6. Let S be the vertex-simple directed path formed by P and C without the edge e.
7. Let Δ be the set of dangling subgraphs of B with respect to S.
8. Let K be the set of the associated dangling edges.
9. **return** the tuple (S, Δ, K).

end.

FIG. 4. *A procedure for splitting a heavy dangling subgraph.*

THEOREM 5.11. *Let B be a bubble graph of size n. Let r be the specified root of B. Then a depth-first search tree of B rooted at r can be computed in $O(\log^5 n)$ time with $n/\log n$ processors on a deterministic* Arbitrary-CRCW PRAM.

Proof. The computation is done by the procedure in Fig. 6. This theorem then follows directly from Lemmas 5.4 and 5.10, and the fact that by the output property of ComputePartialTree, the depth of recursion of ComputeDFSTree is $O(\log n)$. □

5.8. Parallel depth-first search in strong graphs. The next theorem states the main result of this paper.

THEOREM 5.12. *Let G be a strong graph of size n. Let r be a vertex in G. Then a depth-first search spanning tree of G rooted at r can be computed in $O(\log^5 n)$ time with $n/\log n$ processors on a deterministic* Arbitrary-CRCW PRAM.

Proof. The external face of G can be changed so that r is a boundary vertex on that face. Then, by the strong connectivity of G, it is a bubble graph rooted at r. Therefore, this theorem follows from Theorem 5.11 and the fact that the external face can be changed in $O(\log n)$ time on $n/\log n$ processors using list ranking [5], [9], [13] and prefix computation [23], [24]. □

Appendix. All graphs have cycle separators. The following discussion uses depth-first search trees to compute graph separators [17].

A.1. Path and cycle separators of weighted graphs. A *weighted* graph is one with nonnegative vertex weights. To avoid triviality, assume that at least one vertex has a positive weight.

Procedure ComputePartialTree
Input: a bubble graph B with m vertices.
Output:
1. a partial depth-first search tree T of B such that all dangling subgraphs are $2m/3$-light;
2. the set Ω of the dangling subgraphs of B with respect to T.
3. the set L of the associated dangling edges.

begin
1. $(T, \Omega, L) \leftarrow$ SplitHeavyDSG(m,B).
2. **while** some $B' \in \Omega$ is $2m/3$-heavy **do**
   begin
   2-1. Delete B' from Ω.
   2-2. Delete the dangling edge of B' from L.
   2-3. Add the above edge to T to form a larger tree.
   2-4. $(S, \Delta, K) \leftarrow$ SplitHeavyDSG(m,B').
   2-5. Add S to T to form a larger tree.
   2-6. Add Δ to Ω.
   2-7. Add K to L.
   end.
3. **return** the tuple (T, Ω, L).

end.

FIG. 5. *A procedure for computing a partial depth-first search tree.*

Procedure ComputeDFSTree
Input: a bubble graph B with at least two vertices.
Output: a depth-first search tree T of B.
begin
1. $(T, \Omega, L) \leftarrow$ ComputePartialTree(B).
2. Add L to T to form a larger tree.
3. **for** each $B' \in \Omega$ with at least two vertices **do**
   begin
   3-1. $T' \leftarrow$ ComputeDFSTree(B').
   3-2. Add T' to T to form a larger tree.
   end.
4. **return** T.

end.

FIG. 6. *A procedure for computing a depth-first search tree.*

Let G be a weighted directed graph. Let H be a subgraph or vertex subset of G. Let $G - H$ be the subgraph obtained by removing the vertices in H and their incident edges.

Let $\mathcal{W}(H)$ be the total weight of H. The set H is called *heavy* for G if $\mathcal{W}(H) > \mathcal{W}(G)/2$.

Remark. The threshold for heaviness here is different from that in §4.1, which is 2/3.

A *separator* S of G is a vertex subset such that no strong component of $G - S$ is heavy for G. A *cycle* (respectively, *path*) separator is a vertex-simple directed cycle (respectively, path) such that its vertices form a separator.

For technical uniformity, a vertex is considered a trivial cycle. Thus, if a vertex forms a separator, it is a cycle separator. The empty set is considered both a trivial cycle and a trivial path. Thus, if the empty set forms a separator, it is a cycle separator as well as a path separator.

A.2. Computing path separators from depth-first search trees. The next theorem can be applied to a weighted undirected graph by substituting each undirected edge with a pair of directed edges.

THEOREM A.1. *Every weighted directed graph has a path separator.*

Proof. Let G be a weighted directed graph. Without loss of generality, assume that G is strongly connected. Otherwise, replace G with its maximum-weight strong component. Every path separator of that component is also one for G.

A path separator P for G is constructed as follows. Let T be a depth-first search spanning tree of G rooted at an arbitrary vertex r. Let $z_1, \ldots, z_n$ be the vertices of G in the corresponding depth-first search postorder. Let p be the smallest index with $\mathcal{W}(z_1)+\cdots+\mathcal{W}(z_p) \geq \mathcal{W}(G)/2$. Then, $\mathcal{W}(z_1) + \cdots + \mathcal{W}(z_{p-1}) < \mathcal{W}(G)/2$ and $\mathcal{W}(z_{p+1}) + \cdots + \mathcal{W}(z_n) \leq \mathcal{W}(G)/2$.

Let P be the tree path in T from r to z_p. Let $G_L = \{z_1, \ldots, z_{p-1}\}$. Let $G_R = G-(G_L \cup P)$. Note that $\mathcal{W}(G_L) < \mathcal{W}(G)/2$ and $\mathcal{W}(G_R) \leq \mathcal{W}(G)/2$.

P is shown to be a path separator as follows. Draw G on a plane in such a way that for all postorder indices i and j with $i > j$, the vertex z_i is either to the right of z_j or is an ancestor of z_j in T [4].

The vertex z_p either is an ancestor of or is to the right of every vertex in G_L because p is greater than the postorder indices of all vertices in G_L. Also, every vertex in G_R is to the right of z_p because the postorder indices of vertices in G_R are all greater than p and because P consists of z_p and all its ancestors. Therefore, every vertex in G_R is to the right of every vertex in G_L.

Because in depth-first search no edge points from left to right, every strong component of $G - P$ is either entirely in G_L or entirely in G_R. Thus, the weight of a strong component of $G - P$ is at most $\mathcal{W}(G_L)$ or $\mathcal{W}(G_R)$. □

A.3. Computing cycle separators from path separators. The next theorem can also be applied to a weighted undirected graph by edge substitution. Note that a separator obtained by the theorem actually consists of one of the following: no vertex, a single vertex, or at least three vertices. Thus it does not degenerate into an undirected edge after edge substitution is undone.

THEOREM A.2. *Every weighted directed graph has a cycle separator.*

Proof. Let G be a weighted directed graph. Let $P = u_1, \ldots, u_p$ be a path separator of G obtained by Theorem A.1.

P is converted into a cycle separator as follows. Let s be the largest index such that some strong component Z_s of $G - \{u_1, \ldots, u_{s-1}\}$ is heavy for G. Then $u_s \in Z_s$ because the path $u_1, \ldots, u_s$ is still a separator. If s does not exist, then the empty set is a trivial cycle separator. Otherwise, continue the conversion and let $P' = u_1, \ldots, u_s$.

Let t be the smallest index such that a strong component Z_t of $G - \{u_{t+1}, \ldots, u_s\}$ is heavy for G. Then $u_t \in Z_t$ because the path $u_t, \ldots, u_s$ is still a separator.

There are two cases based on whether $s = t$ or not. If $s = t$, then u_s is a trivial cycle separator. Otherwise, $u_t \notin Z_s$ and $u_s \notin Z_t$. Because Z_s and Z_t are heavy for G and are strongly connected, $Z_s \cap Z_t$ contains a vertex z such that there is a vertex-simple directed path

Q from u_s via $Z_s - Z_t$ to z and then via $Z_t - Z_s$ to u_t. Because $Z_s \cup Z_t$ contains none of $u_{t+1}, \ldots, u_{s-1}$, the path Q and the path $u_t, \ldots, u_s$ form a vertex-simple directed cycle with at least three vertices. This cycle is a separator because the path $u_t, \ldots, u_s$ is a separator. □

Acknowledgement. The author wishes to thank Subhrajit Bhattacharya for helpful discussions and the anonymous referee for extremely thorough and helpful comments.

REFERENCES

[1] K. ABRAHAMSON, N. DADOUN, D. G. KIRKPATRICK, AND T. PRZYTYCKA, *A simple tree contraction algorithm*, J. Algorithms, 10 (1989), pp. 287–302.

[2] A. AGGARWAL AND R. J. ANDERSON, *A random NC algorithm for depth first search*, Combinatorica, 8 (1988), pp. 1–12.

[3] A. AGGARWAL, R. J. ANDERSON, AND M. Y. KAO, *Parallel depth-first search in general directed graphs*, SIAM J. Comput., 19 (1990), pp. 397–409.

[4] A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[5] R. J. ANDERSON AND G. L. MILLER, *Deterministic parallel list ranking*, Algorithmica, 6 (1991), pp. 859–868.

[6] C. BERGE, *Graphs*, North-Holland, second ed., New York, 1985.

[7] B. BOLLOBÁS, *Graph Theory*, Springer-Verlag, New York, 1979.

[8] R. COLE AND U. VISHKIN, *The accelerated centroid decomposition technique for optimal tree evaluation in logarithmic time*, Algorithmica, 3 (1988), pp. 329–346.

[9] ———, *Faster optimal prefix sums and list ranking*, Inform. and Comput., 81 (1989), pp. 334–352.

[10] H. GAZIT, G. L. MILLER, AND S. H. TENG, *Optimal tree contraction in the EREW model*, in Concurrent Computations: Algorithms, Architecture, and Technology, S. T. and B. W. Dickinson and S. Schwartz, eds., Plenum, New York, 1988, pp. 139–156.

[11] A. M. GIBBONS AND W. RYTTER, *An optimal parallel algorithm for dynamic expression evaluation and its applications*, Inform. and Comput., 81 (1989), pp. 32–45.

[12] T. HAGERUP, *Optimal parallel algorithms on planar graphs*, Inform. and Comput., 84 (1990), pp. 71–96.

[13] Y. HAN, *An optimal linked list prefix algorithm on a local memory computer*, IEEE Trans. Comput., 40 (1991), pp. 1149–1153.

[14] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[15] X. HE AND Y. YESHA, *A nearly optimal parallel algorithm for constructing depth first spanning trees in planar graphs*, SIAM J. Comput., 17 (1988), pp. 486–491.

[16] J. JA'JA AND S. KOSARAJU, *Parallel algorithms for planar graphs and related problems*, IEEE Trans. Circuits Systems, 35 (1988), pp. 304–311.

[17] M. Y. KAO, *All graphs have cycle separators and planar directed depth-first search is in DNC*, in Lecture Notes in Comput. Sci. 319: VLSI Algorithms and Architectures, the 3rd Aegean Workshop on Computing, J. H. Reif, ed., Springer-Verlag, New York, 1988, pp. 53–63.

[18] ———, *Linear-processor* NC *algorithms for planar directed graphs* I: *Strongly connected components*, SIAM J. Comput., 22 (1993), pp. 431–459.

[19] M. Y. KAO AND P. N. KLEIN, *Towards overcoming the transitive-closure bottleneck: Efficient parallel algorithms for planar digraphs*, in Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, 1990, pp. 181–192.

[20] M. Y. KAO AND G. E. SHANNON, *Linear-processor NC algorithms for planar directed graphs* II: *Directed spanning trees*, SIAM J. Comput., 22 (1993), pp. 460–481.

[21] M. Y. KAO, S. H. TENG, AND K. TOYAMA, *An optimal parallel algorithm for planar cycle separators*, Algorithmica, to appear.

[22] S. R. KOSARAJU AND A. L. DELCHER, *Optimal parallel evaluation of tree-structured computations by raking*, in Lecture Notes in Comput. Sci. 319: VLSI Algorithms and Architectures, the 3rd Aegean Workshop on Computing, J. H. Reif, ed., Springer-Verlag, New York, 1988, pp. 101–110.

[23] C. P. KRUSKAL, L. RUDOLPH, AND M. SNIR, *The power of parallel prefix*, IEEE Trans. Comput., C-34 (1985), pp. 965–968.

[24] R. E. LADNER AND M. J. FISCHER, *Parallel prefix computation*, J. Assoc. Comput. Mach., 27 (1980), pp. 831–838.

[25] R. LIPTON AND R. E. TARJAN, *A separator theorem for planar graphs*, SIAM J. Discrete Meth., 36 (1979), pp. 177–189.

[26] G. L. MILLER, *Finding small simple cycle separators for 2-connected planar graphs*, J. Comput. System Sci., 32 (1986), pp. 265–279.

[27] G. L. MILLER AND J. H. REIF, *Parallel tree contraction, part* 1: *Fundamentals*, in Advances in Computing Research: Randomness and Computation, S. Micali, ed., vol. 5, JAI Press, Greenwich, CT, 1989, pp. 47–72.

[28] ———, *Parallel tree contraction part* 2: *Further applications*, SIAM J. Comput., 20 (1991), pp. 1128–1147.

[29] V. RAMACHANDRAN AND J. H. REIF, *An optimal parallel algorithm for graph planarity*, in Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science, 1989, pp. 282–287.

[30] J. H. REIF, *Depth-first search is inherently sequential*, Inform. Process. Lett., 20 (1985), pp. 229–234.

[31] G. E. SHANNON, *A linear-processor algorithm for depth-first search in planar graphs*, Inform. Process. Lett., 29 (1988), pp. 119–123.

[32] J. R. SMITH, *Parallel algorithms for depth-first search* I. *Planar graphs*, SIAM J. Comput., 15 (1986), pp. 814–830.

[33] R. TAMASSIA AND J. S. VITTER, *Parallel transitive closure and point location in planar structures*, SIAM J. Comput., 20 (1991), pp. 708–725.

[34] R. E. TARJAN, *Depth-first search and linear graph algorithms*, SIAM J. Comput., 1 (1972), pp. 146–160.

[35] W. TUTTE, *Graph Theory*, Encyclopedia of Mathematics and its Applications, vol. 21, Addison-Wesley, Reading, MA, 1984.

SIAM J. COMPUT.
Vol. 24, No. 1, pp. 63–77, February 1995

004

FAST PARALLEL COMPUTATION OF THE POLYNOMIAL REMAINDER SEQUENCE VIA BÉZOUT AND HANKEL MATRICES*

DARIO BINI† AND LUCA GEMIGNANI‡

Abstract. If $u(x)$ and $v(x)$ are polynomials of degree n and m, respectively, $m < n$, all the coefficients of the polynomials generated by the Euclidean scheme applied to $u(x)$ and $v(x)$ can be computed by using $O(\log^3 n)$ parallel arithmetic steps and $n^2/\log n$ processors over any field of characteristic 0 supporting FFT (Fast Fourier Transform). If the field does not support FFT the number of processors is increased by a factor of $\log\log n$; if the field does not allow division by $n!$ the number of processors is increased by a factor of n. This result is obtained by reducing the Euclidean scheme to computing the block triangular factorization of the Bezout matrix associated with $u(x)$ and $v(x)$. This approach is also extended to the evaluation of polynomial gcd (greatest common divisor) over any field of constants in $O(\log^2 n)$ steps with the same number of processors.

Key words. Euclidean scheme, greatest common divisor, Hankel and Bézout matrices, computational complexity, parallel algorithms

AMS subject classifications. 68Q25, 65Y05

1. Introduction.

Let

$$u(x) = \sum_{i=0}^{n} u_i x^i, \qquad v(x) = \sum_{i=0}^{n-1} v_i x^i$$

be two polynomials with coefficients in a field $\mathbf{F}$. The Euclidean scheme, applied to $u(x)$ and $v(x)$, generates a sequence of polynomials $r_i(x)$, $q_i(x)$, such that

$$\begin{aligned} &r_0(x) = u(x), \qquad r_1(x) = v(x) \\ &r_{i-1}(x) = r_i(x)q_i(x) - r_{i+1}(x), \qquad i = 1, \ldots, L, \end{aligned} \tag{1}$$

where $-r_{i+1}(x)$ is the remainder of the division of $r_{i-1}(x)$ and $r_i(x)$ and $r_L(x)$ is the greatest common divisor (gcd) of $u(x)$ and $v(x)$. If $\deg(r_i(x)) = n - i$, $i = 0, \ldots, n$, the Euclidean scheme is carried out in n steps and the quotients $q_i(x)$ are linear polynomials. We refer to this situation as the *normal case*.

The computation of the coefficients of the polynomials in (1) can be performed in $O(n^2)$ arithmetic operations by means of the synthetic division algorithm. Moreover, in the normal case this computation is asymptotically optimal since all the $n(n-1)/2$ coefficients of the polynomials $r_i(x)$ cannot be computed with less than $n(n-1)/2$ arithmetic operations.

The first polylogarithmic parallel solution to the Euclidean scheme computation was given in [7] (hereafter we assume the customary arithmetic PRAM model of parallel computation, where in each step each processor performs at most an arithmetic operation). The algorithm of [7], based on the theory of subresultants [8], [9], reduces the problem to computing n^2 determinants of size $O(n)$ in $O(\log^2 n)$ steps with $O(n^{\omega+2.5-\epsilon})$ processors, where $\omega < 2.38$ is the exponent of matrix multiplication complexity. In [18] the number of processors is reduced to $O(n^{\omega+1})$, but still obtains a processor-inefficient algorithm.

The first processor-efficient algorithm requiring $O(n^2)$ processor has been devised in [3] but only in the normal case and over fields of characteristic 0. This algorithm relies on an

*Received by the editors July 16, 1991; accepted for publication (in revised form) September 21, 1993.

†Dipartimento di Matematica, Università di Pisa via Buonarroti, 56100 Pisa, Italy. The research of this author was partially supported by 60% funds of MURST and by the ESPRIT project POSSO.

‡Dipartimento di Matematica, Università di Parma via M. D'Azeglio 85/A, 43100 Parma, Italy. The research of this author was partially supported by 40% funds of the Progetto Analisi numerica e matematica computazionale of MURST and by G.N.I.M. of C.N.R.

interesting property that relates Hankel matrices (see the definition in §2) to the Euclidean scheme. We prove that in the normal case the Euclidean scheme computation is equivalent to computing the LU factorization of the Hankel matrix $H(u, v)$ generated by the rational function $v(x)/u(x)$. Other interesting properties relating structured matrices, the Euclidean scheme, and the gcd, have been proved in [1], [2], and [15].

In this paper we generalize the results of [3] to any field of constants $\mathbf{F}$ and to the abnormal case. By using suitable results from the theory of partial realizations (see [14] and Proposition 3.1), we prove that the coefficients of all the polynomials in (1) are univocally determined by any block LU factorization of the Hankel matrix $H(u, v)$ associated with $u(x)$ and $v(x)$ (compare Proposition 3.2), or more conveniently, by any block LU factorization of the matrix $JB(u, v)J$ where $B(u, v)$ is the Bézout matrix of $u(x)$ and $v(x)$ and J is the permutation matrix having 1 in the antidiagonal (see the definitions in §2 and relation (13) in §3). More specifically, if $JB(u, v)J = \hat{L}\hat{D}\hat{L}^T$ is such a factorization, then we prove that the entries of suitable columns of $\hat{L}$ coincide with the coefficients of the polynomials $r_i(x)$ of (1). Therefore, for any algorithm for computing the block LU factorization of the matrices $H(u, v)$ and $JB(u, v)J$ there is a corresponding algorithm for computing the coefficients of the polynomials generated by the Euclidean scheme that has the same computational cost.

In order to devise efficient algorithms for the block LU factorization of $JB(u, v)J$, we investigate suitable properties of the Schur complements of the leading principal submatrices of $JB(u, v)J$. In particular we prove that the Schur complement of the ith nonsingular leading principal submatrix of $JB(u, v)J$ is $JB'J$, where $B' = B(r_i, r_{i+1})$ is the Bézout matrix associated with the polynomials $r_i(x)$ and $r_{i+1}(x)$ of (1).

Based on these results, we devise a parallel algorithm for the computation of the block LU factorization of the matrix $JB(u, v)J$ over any field $\mathbf{F}$, in $O(\log^3 n)$ parallel steps with $n^2 p_{\mathbf{F}}(n) q_{\mathbf{F}}(n)/\log n$ processors, where $p_{\mathbf{F}}(n) = 1$ if $\mathbf{F}$ supports FFT, $p_{\mathbf{F}}(n) = \log\log n$ otherwise, $q_{\mathbf{F}}(n) = 1$ if $\mathbf{F}$ allows division by $n!$, $q_{\mathbf{F}}(n) = n$ otherwise.

The algorithm, relying on the divide-and-conquer strategy, is based on the above property of the Schur complements, and on the recently devised techniques for Hankel-like and Toeplitz-like computations [4], [5], [20].

Concerned with the computation of the gcd of $u(x)$ and $v(x)$, we extend the results of [3] to any field of constant and to the case of Bézout matrices. We give a simpler proof of the results of [3] (see the next Proposition 3.6) characterizing gcd in matrix form. Moreover we prove that $\gcd(u(x), v(x))$ is univocally determined by a suitable vector in the kernel of the matrix $JB(u, v)J$ (see Corollary 3.1). Based on these results, we devise a parallel algorithm for the gcd computation in $O(\log^2 n)$ parallel steps with $n^2 p_{\mathbf{F}}(n) q_{\mathbf{F}}(n)/\log n$ processors.

The use of Bézout matrices, rather than Hankel matrices, also leads to better upper bounds to the number of digits needed for the computation in the case of polynomials with integer or rational coefficients (see our comments in §4 and [5] and [13]).

The paper is organized as follows: in §2 we introduce the definitions and the tools needed for our analysis. In §3 we prove the theoretical results relating the Euclidean scheme and gcd computation to Hankel and Bézout matrices. Finally, in §4 we devise the algorithms for the block LU factorization of JBJ (Euclidean scheme computation) and for the computation of a vector in the kernel of JBJ (gcd computation), and perform their parallel cost analysis.

2. Preliminaries. Let $u(x) = \sum_{i=0}^{n} u_i x^i$ and $v(x) = \sum_{i=0}^{m} v_i x^i$ be two polynomials of degree n and m, respectively, where $m < n$. Let

$$f(z, w) = \sum_{i,j=0}^{+\infty} a_{i,j} z^i w^j$$

be a formal power series in the variables z and w. We associate with $f(z, w)$ the semi-infinite matrix $A = (a_{i,j})$, $i, j = 0, 1, \ldots$, and, for any natural n the $n \times n$ matrix $A_n = (a_{i,j})$, $i, j = 0, \ldots, n-1$, the *finite section* of A. We refer to $f(w, z)$ as to the *generating function* of A or of A_n. It is easy to check that

$$\frac{zu(z) - wu(w)}{z - w}$$

is the generating function of a *Hankel matrix*, that is, a matrix whose (i, j) entry is a function of $i + j$, $i, j = 0, 1, \ldots, n-1$.

Let Z be the *down-shift* matrix that has entries $z_{i,j} = 1$ if $i = j + 1$, $z_{i,j} = 0$ otherwise. Z is a special *Toeplitz matrix*. (A matrix is Toeplitz if its (i, j) entry is a function of $i - j$.)

The $n \times n$ matrix F_u obtained by replacing the last row of Z^T by the vector $(-u_0/u_n, \ldots, -u_{n-1}/u_n)^T$ is the *Frobenius (companion) matrix* associated with $u(x)$. We denote by $T(a_0, \ldots, a_{n-1})$ the lower triangular Toeplitz matrix having entries a_{i-j}, $i \geq j$.

The expression

$$\frac{u(z)v(w) - v(z)u(w)}{z - w}, \tag{2}$$

which is easily verified to be a polynomial in w and z, is called the *Bézoutian* of $u(x)$ and $v(x)$. The Bézoutian is a rather special bilinear form which appears in the context of the theory of equations, in stability theory, and in the classical elimination theory [16], [19], [11], [23]. The $n \times n$ matrix $B(u, v)$, whose generating function is (2), is called the *Bézout matrix* or, more simply the *Bézoutian* of $u(x)$ and $v(x)$. The following matrix representation of the Bézoutian [19] can easily be proved:

$$B(u, v) = -JT(v_n, \ldots, v_1)T^T(u_0, \ldots, u_{n-1}) + JT(u_n, \ldots, u_1)T^T(v_0, \ldots, v_{n-1}), \tag{3}$$

where J is the permutation matrix having 1 in the antidiagonal, i.e., its (i, j) entry is 1 for $j = n - i - 1$, $i, j = 0, \ldots, n-1$.

Remark 2.1 Given the mth row and the last row of $B(u, v)$, together with the entry $(0, 0)$ lying in the first row and in the first column of $B(u, v)$, it is possible to compute the coefficients of the polynomials $u_n v(x)$ and $u(x)/u_n$. In fact, if $m < n$, the last row of $B(u, v)$ is $u_n[v_0, \ldots, v_{n-1}]$, which defines the coefficients of $u_n v(x)$. Moreover, the mth row $\mathbf{b}^T$ of $B(u, v)$ is

$$\mathbf{b}^T = -v_m[u_0, \ldots, u_{n-1}] + [u_m, \ldots, u_n, 0, \ldots, 0] \begin{pmatrix} v_0 & \ldots & v_{n-1} \\ & \ddots & \vdots \\ O & & v_0 \end{pmatrix}, \tag{4}$$

and this relation can be used to express the coefficients of $u(x)/u_n$ as linear functions in u_{n-1}. Indeed, assume for simplicity that $u_n = 1$, so that the last row of $B(u, v)$ gives us the value of m and the coefficients $v_0, \ldots, v_m$. Moreover, rewrite (4) as an $n \times n$ system of linear equations:

$$[u_0, \ldots, u_{n-1}] \left(-v_m I_n + \begin{pmatrix} & & O & & O \\ v_0 & \ldots & v_m & & \\ & \ddots & & \ddots & \\ O & & v_0 & \ldots & v_m \end{pmatrix} \right)$$
$$= \mathbf{b}^T - [0, \ldots, 0, v_0, \ldots, v_{m-1}].$$

Remove the last equation of this linear system and consider the triangular Toeplitz system consisting of the last remaining $n-m$ equations. From the latter system, express $u_n, \ldots, u_{n-2}$ in terms of u_{n-1}. Substitute $u_m, \ldots, u_{n-1}$ in the first m equations and express them via u_{n-1}. Since we are given the entry in the first row and in the first column of $B(u, v)$, we may define the remaining unknown value u_{n-1}.

Another property, which directly follows from representation (3), is that the matrix $B(u, 1)$ is an upper triangular Hankel matrix:

$$B(u,1) = \begin{pmatrix} u_1 & u_2 & \cdots & u_n \\ u_2 & \iddots & \iddots & \\ \vdots & \iddots & O & \\ u_n & & & \end{pmatrix}.$$

Let h be a constant independent of n and L_i, U_i^T, $i = 1, \ldots, h$, be lower triangular Toeplitz matrices. Any matrix $A = \sum_{i=1}^{h} L_i U_i$ is called Toeplitz-like, any matrix $B = \sum_{i=1}^{h} J L_i U_i$ is called Hankel-like. From (3) it follows that $B(u, v)$ is a Hankel-like matrix.

Now consider the formal power series

$$\frac{v(x^{-1})}{xu(x^{-1})} = \sum_{i=0}^{+\infty} h_i x^i, \tag{5}$$

whose coefficients are related to the coefficients of $u(x)$ and $v(x)$ by the triangular Toeplitz system of equations

$$\begin{pmatrix} h_0 & & & \\ h_1 & h_0 & & \\ h_2 & h_1 & h_0 & \\ \ddots & \ddots & \ddots & \ddots \end{pmatrix} \begin{pmatrix} u_n \\ u_{n-1} \\ u_{n-2} \\ \vdots \end{pmatrix} = \begin{pmatrix} v_{n-1} \\ v_{n-2} \\ v_{n-3} \\ \vdots \end{pmatrix} \tag{6}$$

where we assume $v_i = 0$ for $i > m$. The above power series defines the $n \times n$ Hankel matrix $H(u, v)$, whose (i, j) entry is h_{i+j}, $i, j = 0, 1, \ldots, n-1$. Relation (6) immediately implies the following.

PROPOSITION 2.1. *For any pair of relatively prime polynomials $u(x)$, $v(x)$ of degree n and m, respectively, $m < n$, the matrix $H(u, v)$ is nonsingular, moreover, for any nonsingular $n \times n$ Hankel matrix H there exists a pair of relatively prime polynomials $u(x)$, $v(x)$, ($u(x)$ monic), of degree n and m, respectively, $m < n$, such that $H = H(u, v)$. The polynomials $u(x)$ and $v(x)$ are related to $H(u, v)$ by the following equation:*

$$H(u,v)(u_0, \ldots, u_{n-1})^T = -u_n(h_n, \ldots, h_{2n-1}),$$
$$(v_{n-1}, \ldots, v_0)^T = T(h_0, \ldots, h_{n-1})(u_n, \ldots, u_1)^T,$$

where h_{2n-1} is any number.

Proof. Observe that the first n equations of (6) allow us to express the coefficients of $v(x)$ in terms of the coefficients of $u(x)$. The remaining n equations constitute a system in the unknowns $u_0, \ldots, u_{n-1}$, whose matrix is $H(u, v)$, with known terms $-u_n(h_n, \ldots, h_{2n-1})^T$. Therefore, if $\det H(u, v) \neq 0$, then there exists unique (up to a scalar factor and once h_{2n-1} has been fixed) solution $u(x)$ to this system, and consequently, there exists unique polynomial $v(x)$. In this case $\gcd(u(x), v(x)) = 1$ otherwise the above linear system would have more than one solution. Vice versa, if $\det H(u, v) = 0$, the above system (with h_{2n-1} fixed) would

have more than one solution, say, $u(x)$, $v(x)$ and $a(x)$, $b(x)$ of degrees t, s, respectively, $s \leq t$. Therefore, from the relation

$$\frac{v(x^{-1})}{xu(x^{-1})} - \frac{b(x^{-1})}{xa(x^{-1})} = 0 \bmod x^{2n}$$

it follows that

$$v(x^{-1})a(x^{-1}) - u(x^{-1})b(x^{-1}) = 0 \bmod x,$$

that is, since the left-hand side is a polynomial in x^{-1} of degree at most $2n-1$ it follows that

$$v(x)a(x) - u(x)b(x) = 0,$$

hence $u(x)$ and $v(x)$ have a common nonconstant divisor. □

Remark 2.2. Let $H = H(u, v)$ be an $n \times n$ nonsingular Hankel matrix. Then $m = \deg v = n - k - 1$ if and only if $h_i = 0$, $i = 0, 1, \ldots, k-1$, $h_k \neq 0$, that is, if and only if $\det H_i = 0$, $i = 1, \ldots, k$, where H_i is the $i \times i$ leading principal submatrix of H.

We recall a classical result relating the matrices $H(u, v)$ and $B(u, v)$ [17], [16], [10].

PROPOSITION 2.2. *Let F_u be the $n \times n$ Frobenius matrix associated with the polynomial $u(x)$ of degree n. Let $v(x)$ be a polynomial of degree $m < n$. Then we have*

$$B(u, v) = B(u, 1)H(u, v)B(u, 1),$$

$$B(u, v) = B(u, 1)v(F_u),$$

moreover, if $u(x)$ and $v(x)$ are relatively prime then

$$B(u, k)H(u, v) = I,$$

where $v(F_u) = \sum_{i=0}^{m} v_i F_u^i$ and $k(x)$ is the polynomial of degree less than n such that $k(x)v(x) = 1 \bmod u(x)$.

Compare the first and second representations of $B(u, v)$ given in Proposition 2.2. Since $B(u, 1)$ is nonsingular it follows that

$$H(u, v) = (B(u, 1))^{-1} v(F_u^T). \tag{7}$$

If $H(u, v)$ is singular the above relation allows us to characterize the kernel of $H(u, v)$; in fact, the following result holds.

LEMMA 2.1. *Let $u(x)$ and $v(x)$ be two polynomials of degrees n and m, respectively, where $m < n$. Then $v(F_u^T)\mathbf{z} = \mathbf{0}$ if and only if $v(x)z(x) = 0 \bmod u(x)$, where $z(x) = \sum_{i=0}^{n-1} z_i x^i$. Therefore, $v(F^T)$ is singular if and only if $r(x) = \gcd(u(x), v(x))$ is a nonconstant polynomial. Moreover, if $u(x) = w(x)r(x)$, the null space of $v(F^T)$ is $\{\mathbf{z} \in \mathbf{C}^n : z(x) = p(x)w(x)\}$.*

Proof. Since $(F^T)^i \mathbf{e}^{(0)} = \mathbf{e}^{(i)}$, $i = 0, \ldots, n-1$ (where $\mathbf{e}^{(i)}$ denotes the ith column of the $n \times n$ identity matrix), the first column of $v(F^T)$ is $(v_0, v_1, \ldots, v_m, 0, \ldots, 0)^T$. Therefore, $v(F^T)\mathbf{z} = v(F^T)z(F^T)\mathbf{e}^{(0)} = s(F^T)\mathbf{e}^{(0)}$, where $s(x) = v(x)z(x) \bmod u(x)$ is a polynomial of degree less than n. Hence, $v(F^T)\mathbf{z} = \mathbf{0}$ if and only if $v(x)z(x) = 0 \bmod u(x)$. In particular, $z(x)$ must be a multiple of $w(x)$, i.e., there exists a polynomial $p(x)$ such that $z(x) = p(x)w(x)$. □

The above lemma will be used in order to characterize the polynomial gcd in terms of the Hankel matrix (Proposition 3.6).

3. Theoretical results. In this section we analyze correlations between the matrices $H(u, v)$, $B(u, v)$, and the coefficients of the polynomials generated by the Euclidean scheme applied to $u(x)$ and $v(x)$.

It is well known that Hankel matrices and their factorizations are related to the polynomial remainder sequences by means of the nonsymmetric Lanczos process for tridiagonalizing general non-Hermitian square matrices. (We refer the reader to [12] and [21] and to the references given there.) Here, we first recall a result of [14] which relates triangular factorization of Hankel matrices to orthogonalization processes. Then we will prove that suitable rows of the matrix $\tilde{L}^{-1}$ such that $H(u, k) = \tilde{L}\tilde{D}\tilde{L}^T$ is a block LDL^T factorization of $H(u, k)$ yield the coefficients of the remainders generated by the Euclidean scheme applied to $u(x)$ and $v(x)$. Here the polynomial $k(x)$ is the solution of the congruence $v(x)k(x) = 1 \bmod u(x)$. Then we prove that the matrix $\tilde{L}^{-1}$ can be obtained, bypassing the computation of $k(x)$ and the inversion of the matrix $\tilde{L}$, just by computing the L factor of the block LDL^T factorization of $JB(u, v)J$. Finally, by proving some properties of the Schur complements of the Bézout matrices, we arrive at the complete equivalence between the Euclidean scheme and the block LDL^T factorization of the matrix $JB(u, v)J$. We conclude this section by proving that the coefficients of $\gcd(u(x), v(x))$ can be directly obtained from a suitable vector in the kernel of $JB(u, v)J$.

Let us start by recalling the following basic result of [14].

PROPOSITION 3.1. *Let $H = (h_{i+j})$ be a semi-infinite Hankel matrix and H_n be its $n \times n$ finite section. Suppose that H_n is nonsingular and let $0 = m_0 < m_1 < \cdots < m_L = n$ be integers such that $\det H_{m_i} \neq 0$, $i = 1, \ldots, L$, $\det H_k = 0$, otherwise. Let $\delta_i = m_i - m_{i-1}$. Let $\mathcal{D}_n$ be the class of block diagonal matrices $\operatorname{diag}(T_1, \ldots, T_L)$ where T_i is a $\delta_i \times \delta_i$ lower triangular (with respect to the antidiagonal) Hankel matrix. Then we have the following results*:

(1) *There exists an upper triangular matrix R having unit diagonal entries, such that $R^T H_n R = D$, $D \in \mathcal{D}_n$. In particular one can choose*

$$R = \tilde{R} = (\mathbf{q}_0, Z\mathbf{q}_0, \ldots, Z^{\delta_1 - 1}\mathbf{q}_0, \mathbf{q}_1, Z\mathbf{q}_1, \ldots, Z^{\delta_2 - 1}\mathbf{q}_1, \ldots\ldots, Z^{\delta_L - 1}\mathbf{q}_{L-1}),$$

where Z is the down-shift matrix, $\mathbf{q}_0 = \mathbf{e}^{(0)}$, $\mathbf{q}_i^T = (\tilde{\mathbf{q}}_i^T, 1, \mathbf{0}^T)$, $\tilde{\mathbf{q}}_i = -H_{m_i}^{-1}(h_{m_i}, \ldots, h_{2m_i - 1})^T$, $i = 1, \ldots, L - 1$;

(2) *any upper triangular matrix $R = (\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{n-1})$, such that $R^T H_n R \in \mathcal{D}_n$, is such that $\mathbf{r}_{\delta_i} = \mathbf{q}_i$, where we assume $\delta_0 = 0$;*

(3) *set $Q_i(x) = (1, x, \ldots, x^{n-1})\mathbf{q}_i$, $i = 0, 1, \ldots, L$, where*

$$\mathbf{q}_L = -H_n^{-1} \begin{pmatrix} h_{n+1} \\ \vdots \\ h_{2n} \end{pmatrix},$$

and we have

$$\begin{aligned} &Q_i(x) = c_i(x)Q_{i-1}(x) - \theta_{i-1}Q_{i-2}(x), \qquad i = 1, \ldots, L, \\ &Q_0(x) = 1, \qquad Q_{-1}(x) = 0, \end{aligned}$$

where $c_i(x)$ are monic polynomials of degree δ_i and θ_{i-1} are constants $i = 1, \ldots, L$.

It is easy to show [12], [21] that the polynomials $Q_i(x)$, $i = 0, \ldots, L$, are orthogonal with respect to the symmetric bilinear form $\langle b(x), c(x)\rangle = \mathbf{b}^T H_n \mathbf{c}$, $b(x) = (1, x, \ldots, x^{n-1})\mathbf{b}$, $c(x) = (1, x, \ldots, x^{n-1})\mathbf{c}$ on the linear space of polynomials of degree less than n. This is the basic relationship between Hankel matrices and systems of orthogonal polynomials generated by three-term recurrence relations.

It is interesting to point out that the block triangular factorization of Proposition 3.1 is not generally unique.

Applying Proposition 3.1 to the matrix $H_n = H(u, v)$ it is possible to characterize the vectors $\mathbf{q}_i$ and, consequently, the polynomials $Q_i(x)$. We have, in fact, the following result.

PROPOSITION 3.2. *Let $u(x)$ and $v(x)$ be monic polynomials of degrees n and m, respectively, such that $n > m$, $\gcd(u(x), v(x)) = 1$. Then for the monic polynomials $Q_L(x)$, $Q_{L-1}(x)$ defined in Proposition* 3.1 *for $H = H(u, v)$, we have*

$$Q_L(x) = u(x), \qquad Q_{L-1}(x) = k^*(x),$$

where $k^*(x)$ is the monic polynomial associated with $k(x)$ (i.e., $k^*(x) = ck(x)$ for a nonzero constant c), and $k(x)v(x) = 1 \bmod u(x)$. Hence, the polynomials $Q_i(x)$, $i = 0, 1, \ldots, L$ are the monic polynomials associated with the polynomials obtained by applying the Euclidean scheme to $u(x)$ and $k(x)$.

Proof. For Proposition 2.2 we have

$$H(u, v)\mathbf{k} = \mathbf{e}^{(n-1)}, \quad \mathbf{k} = (k_0, \ldots, k_{n-1})^T, \quad k_i = 0, \ i > \deg\ k(x).$$

In particular, since

$$H_i(u, v)(k_0, \ldots, k_{i-1})^T = \mathbf{0}, \ i \geq 1 + \deg\ k(x) \tag{8}$$

(this condition does not apply if $\deg\ k(x) = n - 1$), we have

$$\det H_i = 0, \quad i = 1 + \deg\ k(x), \ldots, n - 1, \quad \det H_i \neq 0, \quad i = \deg\ k(x)$$

(for the uniqueness of $k(x)$), hence $\deg\ k(x) = m_{L-1}$; moreover,

$$H_{m_{L-1}}(k_0, \ldots, k_{m_{L-1}-1})^T = -k_{m_{L-1}}(h_{m_{L-1}}, \ldots, h_{2m_{L-1}-1})^T. \tag{9}$$

From parts (1) and (3) of Proposition 3.1 it follows that $Q_{L-1} = k^*(x)$. Now, let

$$\tilde{u}(x) = (x - \alpha)u(x), \qquad \tilde{v}(x) = (x - \alpha)v(x), \quad \alpha \in \mathbf{F}.$$

For Lemma 2.1 the vector $\mathbf{u} = (u_0, u_1, \ldots u_{n-1}, 1)^T$ generates the kernel of the matrix $H(\tilde{u}, \tilde{v})$. Thus, from $H(\tilde{u}, \tilde{v})\mathbf{u} = \mathbf{0}$ we obtain that

$$H(u, v)(u_0, \ldots, u_{n-1})^T = -(h_{n+1}, \ldots, h_{2n})^T,$$

that is, $u(x) = Q_L(x)$. □

The above results can be restated in the following way: Given two monic and relatively prime polynomials $u(x)$ and $v(x)$, the coefficients of the polynomials generated by the Euclidean scheme applied to $u(x)$ and $v(x)$ are given by suitable rows of the matrix $\tilde{L}^{-1}$, where $H(u, k) = \tilde{L}\tilde{D}\tilde{L}^T$ is a block triangular factorization of $H(u, k)$ and the polynomial $k(x)$ solves the congruence $v(x)k(x) = 1 \bmod u(x)$. It is easy to show that if $H(u, v) = LDL^T$ is a block triangular factorization of $H(u, v)$, then

$$H(u, k) = \tilde{L}\tilde{D}\tilde{L}^T, \quad \text{where } \tilde{L}^{-1} = JL^TB(u, 1), \quad \tilde{D}^{-1} = JDJ$$

is a block triangular factorization of $H(u, k)$. Therefore, the computation of the coefficients of all the polynomials generated in the Euclidean scheme, applied to the polynomials $u(x)$ and $v(x)$, is reduced to computing the block triangular factorization $H(u, v) = LDL^T$ and to

matrix-vector products with a lower triangular Hankel matrix. This way, unlike [3], we avoid the computation of the polynomial $k(x)$ and the inversion of the matrix L. Moreover, since

$$JB(u, v)J = JB(u, 1)H(u, v)B(u, 1)J, \tag{10}$$

the block triangular factorization of $H(u, v)$ can be obtained from the block triangular factorization of $JB(u, v)J = \hat{L}\hat{D}\hat{L}^T$. Indeed, we may choose $\tilde{L}^{-1} = J\hat{L}^T J$ (recall that the block LU factorization is not unique), therefore suitable columns of $\hat{L}$ give the coefficients of the polynomials generated by the Euclidean scheme.

Observe that the entries of $H(u, v)$ may grow exponentially with n, whereas the entries of $JB(u, v)J$ are bounded by $2n\mu\nu$, where μ and ν are upper bounds to the moduli of the coefficients of $u(x)$ and $v(x)$, respectively. Moreover, if $u(x)$ and $v(x)$ have integer coefficients then $JB(u, v)J$ has integer entries. This makes computing the block triangular factorization of $JB(u, v)J$ more convenient than computing the block triangular factorization of $H(u, v)$, and leads to an algorithm having a lower Boolean cost (see [5] and [13]).

The results proved so far imply the reduction of the Euclidean scheme computation to computing the block LDL^T factorization of $JB(u, v)J$. In order to devise efficient parallel algorithms for the computation of this factorization we will investigate suitable properties of the Schur complements of the nonsingular leading principal submatrices of $JB(u, v)J$.

Given the matrix

$$\begin{pmatrix} M & P \\ Q & R \end{pmatrix},$$

where M and R are square matrices and M is nonsingular, the Schur complement of M in A is the matrix $S = R - QM^{-1}P$. It is well known that if A is nonsingular then S is nonsingular and S^{-1} is the right lower corner submatrix of A^{-1}.

Now denote $(H(u, v))_i$, $(JB(u, v)J)_i$ the $i \times i$ leading principal submatrix of $H(u, v)$ and $JB(u, v)J$, respectively, and observe that in the hypotesis of Proposition 3.2, from Remark 2.2 we have

$$\det(H(u, v))_i = 0, \quad i = 1, \ldots, n - m - 1, \quad \det(H(u, v))_{n-m} \neq 0.$$

Therefore from (10) it follows that

$$\det(JB(u, v)J)_i = 0, \quad i = 1, \ldots, n - m - 1, \quad \det(JB(u, v)J)_{n-m} \neq 0. \tag{11}$$

We recall the following result of [22] concerning the generating function of the Schur complements.

LEMMA 3.1. *If Q is an $n \times n$ matrix having generating function*

$$\frac{a(z)b(w) - b(z)a(w)}{z - w}$$

and $\det Q_i = 0$, $i = 1, \ldots, t - 1$, $\det Q_t \neq 0$, *then the Schur complement of Q_t in Q has generating function*

$$\frac{a_t(z)b_t(w) - b_t(z)a_t(w)}{z - w},$$

where $a_t(z) = z^{-t}a(z)$ and $b_t(z)$ is defined by the recurrence

$$b_0(z) = b(z),$$
$$zb_{j+1}(z) = b_j(z) - a_t(z)\frac{b_j(0)}{a_t(0)}.$$

Applying Lemma 3.1 we prove the following basic result relating the Schur complement of the first nonsingular leading principal submatrix of $JB(u, v)J$ to the remainder $s(x)$ of the division of $u(x)$ and $v(x)$.

PROPOSITION 3.3. *Let S be the Schur complement of $(JB(u, v)J)_{n-m}$ in $JB(u, v)J$, then we have*

$$S = JB(v, s)J,$$

where $u(x) = q(x)v(x) - s(x)$, deg $s(x) < m$.

Proof. The generating function of $JB(u, v)J$ is easily verified to be

$$\frac{-\tilde{u}(z)\tilde{v}(w)w^{n-m} + \tilde{u}(w)\tilde{v}(z)z^{n-m}}{z - w},$$

where $\tilde{u}(z) = z^n u(z^{-1})$ and $\tilde{v}(z) = z^m v(z^{-1})$ are the reversed polynomials of $u(z)$, $v(z)$, respectively. Therefore, for Lemma 3.1, the generating function of the Schur complement S is given by

$$\frac{v_{n-m}(z)u_{n-m}(w) - u_{n-m}(z)v_{n-m}(w)}{z - w},$$

where $v_{n-m}(z) = \tilde{v}(z)$, and

$$\begin{aligned} &u_0(z) = \tilde{u}(z), \\ &zu_{j+1}(z) = u_j(z) - \tilde{v}(z)\tfrac{u_j(0)}{\tilde{v}(0)}, \quad j = 0, \ldots, n - m - 1. \end{aligned} \tag{12}$$

Now observe that the polynomials $u_j(z)$ in (12) have degrees $n - j$, $j = 0, \ldots, n - m$, and that $z^{n-j}u_j(z^{-1})$ are the polynomials obtained at the stage j of the synthetic polynomial division algorithm applied to $u(x)$ and $v(x)$, so that

$$u_{n-m+1}(z) = -\tilde{s}(z)z^{m-\deg s-1}, \quad u(z) = q(z)v(z) - s(z), \quad \deg s(z) < m.$$

Therefore

$$u_{n-m}(z) = z^{m-\deg s}\tilde{s}(z) + \beta\tilde{v}(z), \quad \beta = \frac{\tilde{u}_{n-m}(0)}{\tilde{v}(0)}.$$

Hence, the generating function of S is

$$\frac{-\tilde{v}(z)\tilde{s}(w)w^{m-\deg s} + \tilde{v}(w)\tilde{s}(z)z^{m-\deg s}}{z - w}$$

so that $S = JB(v, s)J$. □

Now consider the Schur complement S_k and S_{h+k} of A_k, A_{h+k}, respectively, in A, where A, A_k, A_{h+k}, are nonsingular matrices and A_k, A_{h+k} are the leading principal submatrices of A of dimension $k \times k$, $(h + k) \times (h + k)$, respectively. Since S_k^{-1} and S_{h+k}^{-1} are the right lower corner submatrices of A^{-1} of sizes $k \times k$ and $(h + k) \times (h + k)$, respectively, we have that S_{h+k} is the Schur complement of $(S_k)_h$ in S_k. Therefore we may inductively apply (11) and Proposition 3.3, obtaining the following proposition.

PROPOSITION 3.4. *Let $u(x)$ and $v(x)$ be monic polynomials of degrees n and m, respectively, such that $n > m$, $\gcd(u(x), v(x)) = 1$. Let $m_1 < \cdots < m_L = n$ be integers such that*

$$\det H_{m_i} \neq 0, \quad i = 1, \ldots, L, \quad \det H_j = 0, \quad j \neq m_i,$$

where the semi-infinite Hankel matrix H is generated by the power series (5). *Then*

$$\det(JB(u,v)J)_{m_i} \neq 0, \quad i = 1,\dots,L, \quad \det(JB(u,v)J)_j = 0, \quad j \neq m_i.$$

Morover, for the Schur complement S_{m_i} of $(JB(u,v)J)_{m_i}$ in $JB(u,v)J$ the following relation holds:

$$S_{m_i} = JB(r_i, r_{i+1})J,$$

where $r_i(x)$, $i = 1,\dots,L$ is the remainder sequence obtained by applying the Euclidean scheme to $u(x)$ and $v(x)$.

Remark 3.1. From the above result it follows that the nonsingular leading principal submatrices of $JB(u,v)J$ have sizes $m_i = n - \deg\ r_i(x)$.

Let $l_i = n - m_i + 1 = \deg\ r_i(x) + 1$ and $P_{m_i} = (JB(u,v)J)_{m_i}$ for $i = 1,\dots,L$. The following result relates the matrices P_{m_i}, $i = 1,\dots,L$ to the polynomials defining the Padé approximants of the power series (5).

PROPOSITION 3.5. *In the hypothesis of Proposition* 3.4, *we have*

$$P_{m_i} = T(u_n,\dots,u_{l_i})T(u^{(i)}_{m_i},\dots,u^{(i)}_1)^{-1}JB(u^{(i)},v^{(i)})JT(u^{(i)}_{m_i},\dots,u^{(i)}_1)^{-T}T(u_n,\dots,u_{l_i})^T,$$

where the polynomials $u^{(i)}(x)$, $v^{(i)}(x)$ are such that

$$H(u^{(i)},v^{(i)}) = (H(u,v))_{m_i}, \quad u^{(i)}(x) = \sum_{r=0}^{m_i} u^{(i)}_r x^r, \quad v^{(i)}(x) = \sum_{r=0}^{m_i-1} v^{(i)}_r x^r.$$

Proof. The proposition follows from the relation

$$P_{m_i} = (JB(u,v)J)_{m_i} = T(u_n\dots,u_{l_i})(H(u,v))_{m_i}T(u_n\dots,u_{l_i})^T$$

in the view of Proposition 2.2 and (10). □

Propositions 3.4 and 3.5 lead to the following block factorization of $JB(u,v)J$:

$$JB(u,v)J = \begin{pmatrix} P_{m_i} & X^T \\ X & W \end{pmatrix} = \begin{pmatrix} I & O \\ XP_{m_i}^{-1} & I \end{pmatrix}\begin{pmatrix} P_{m_i} & O \\ O & S_{m_i} \end{pmatrix}\begin{pmatrix} I & P_{m_i}^{-1}X^T \\ O & I \end{pmatrix}$$

$$= \begin{pmatrix} T & O \\ XP_{m_i}^{-1}T & I \end{pmatrix}\begin{pmatrix} JB(u^{(i)},v^{(i)})J & O \\ O & JB(r_i,r_{i+1})J \end{pmatrix}\begin{pmatrix} T^T & T^TP_{m_i}^{-1}X^T \\ O & I \end{pmatrix}, \tag{13}$$

where

$$S_{m_i} = W - XP_{m_i}^{-1}X^T = JB(r_i,r_{i+1})J, \quad T = T(u_n,\dots,u_{l_i})(B(u^{(i)},1))^{-1}J.$$

Proposition 3.4 and Remark 2.1 allow us to express the coefficient vector $\mathbf{r}_{i+1}$ of the $(i+1)$st remainder $r_{i+1}(x)$ as

$$\mathbf{r}_{i+1} = J(W - XP_{m_i}^{-1}X^T)J\mathbf{e}^{(n-m_i)},$$

that is, $\mathbf{r}_{i+1}$ can be computed by solving an $m_i \times m_i$ system with matrix P_{m_i}. Observe that P_{m_i} is the sum of products of pairs of triangular Hankel and Toeplitz matrices. Matrices having this structure are called Hankel-like matrices and have very strong computational properties [4], [5], [20].

In the case where the polynomials $u(x)$ and $v(x)$ have integer coefficients, the solution of the above Hankel-like system can be performed over the integers by multiplying both members by $\det P_{m_i}$. This way the polynomial $\det P_{m_i} r_{i+1}(x)$ has integer coefficients having moduli $O((2n\mu\nu)^{m_i})$. In the next section we will recursively apply (13) in order to compute the polynomials $r_i(x)$ with a low parallel cost

The assumption $\gcd(u(x), v(x)) = 1$ is no loss of generality. Indeed, it is easy to see that if $\gcd(u(x), v(x)) = r_L(x)$ the Euclidean remainder sequences $\{r_i(x)\}$ and $\{q_i(x)\}$ can be recovered by the polynomial sequences $\{\bar{r}_i(x)\}$ and $\{\bar{q}_i(x)\}$ generated by the Euclidean scheme applied to $u(x)/r_L(x)$ and $v(x)/r_L(x)$.

Let us conclude this section by analyzing the matrix formulation of computing gcd. The algorithm for the gcd computation given in [3] relies on the following characterization of gcd stated in terms of the Hankel matrix $H(u, v)$. Here we propose a different proof that does not make use of the zeros of the polynomials $u(x)$ and $v(x)$ as it is done in [3]; this allows us to extend the computation over any field of constants.

PROPOSITION 3.6. *Let $u(x)$, $v(x)$ be two monic polynomials of degrees n, m, respectively, $m < n$; $r(x) = \gcd(u(x), v(x))$ be a monic polynomial of degree $n - k$, and H_i be the $i \times i$ leading principal submatrix of $H(u, v)$. Then we have*

(a) $\operatorname{rank}(H(u, v)) = k$, $\det H_k \neq 0$, $\det H_i = 0$ *for* $i > k$,

(b) *if* $H_{k+1}\mathbf{w} = \mathbf{0}$, $\mathbf{w} = (w_i)_{i=0,\ldots,k}$, $w_k = 1$, *that is, if*

$$H_k \begin{pmatrix} w_0 \\ \vdots \\ w_{k-1} \end{pmatrix} = - \begin{pmatrix} h_k \\ \vdots \\ h_{2k-1} \end{pmatrix},$$

then

$$u(x) = r(x) \sum_{i=0}^{k} w_i x^i.$$

Proof. Assume that $u(x) = w(x)r(x)$ and $v(x) = t(x)r(x)$, so that $w(x)$ and $t(x)$ are relatively prime. From (7), we obtain that

$$H(u, v)\mathbf{z} = \mathbf{0} \quad \Longleftrightarrow \quad v(F_u^T)\mathbf{z} = \mathbf{0}.$$

So, the problem is reduced to analyzing the null space of $v(F_u^T)$. Now, by the virtue of Lemma 2.1, $v(F_u^T)\mathbf{z} = \mathbf{0}$ holds if and only if the polynomial $z(x) = \sum_{i=0}^{n-1} z_i x^i$ is such that $z(x) = p(x)w(x)$ for some polynomial $p(x)$. Since $\{z(x) : \quad z(x) = p(x)w(x), \deg p(x) \leq n-k-1\}$ is a linear space of dimension $n-k$, we have that rank $H(u, v) =$ rank $v(F^T) = k$. Moreover, by choosing $p(x) = x^j$, $j = 0, \ldots, n-k-1$, we satisfy the relations (a) and (b). □

From Lemma 2.1 and (10) it follows that Proposition 3.6 can be rewritten in terms of the Bézout matrix $B(u, v)$.

COROLLARY 3.1. *In the hypothesis of Proposition* 3.6 *we have*

(a) $\operatorname{rank}(B(u, v)) = k$, $\det(JB(u, v)J)_k \neq 0$, *and* $\det(JB(u, v)J)_i = 0$, $i > k$,

(b) *if* $(JB(u, v)J)_{k+1}\mathbf{y} = \mathbf{0}$, $\mathbf{y} = (y_0, \ldots, y_k)^T$, $y_k = 1$ *then*

$$u(x) = r(x) \sum_{i=0}^{k} w_i x^i,$$

where $\mathbf{w} = (B(u, 1)J)_{k+1}\mathbf{y}$, $\mathbf{w} = (w_0, \ldots, w_k)^T$, *and* $(JB(u, v)J)_i$ *denotes the* $i \times i$ *leading principal submatrix of* $JB(u, v)J$.

In the next section we will devise an algorithm for computing $\gcd(u(x), v(x))$ based on the above corollary.

4. The algorithms and their cost analysis. By using the results of §3 we devise new algorithms for the computation of polynomial gcd and of the coefficients of the polynomials generated in the Euclidean scheme.

We make use of the following computational results some of which have been recently proved in [4], [5], [6], [20]. In order to express the computational cost we use the functions $p_{\mathbf{F}}(n)$ and $q_{\mathbf{F}}(n)$, defined by $p_{\mathbf{F}}(n) = 1$ if the field $\mathbf{F}$ supports FFT, $p_{\mathbf{F}}(n) = \log\log n$ otherwise, $q_{\mathbf{F}}(n) = 1$ if $\mathbf{F}$ allows division by $n!$, $q_{\mathbf{F}}(n) = n$ otherwise.

Let $L_i, U_i^T, i = 1, \ldots, h$, be lower triangular Toeplitz matrices with entries in the field $\mathbf{F}$, where h is a constant independent of n. Consider the Hankel-like matrix $A = \sum_{i=1}^{h} JL_iU_i$. The matrix A is determined by its *displacement generator* made up by the set of vectors which are the first columns of the matrices L_i and U_i^T.

PROPOSITION 4.1. *For the Hankel-like matrix A the following computational results hold:*

- *The computation of* $r = \operatorname{rank}(A)$ *can be performed in* $O(\log^2 n)$ *parallel steps with* $n^2 p_{\mathbf{F}}(n) q_{\mathbf{F}}(n)/\log n$ *processors assuming the nonsingularity of the* $r \times r$ *leading principal submatrix of A.*
- *If A is nonsingular the solution of the linear system* $A\mathbf{x} = \mathbf{b}$ *can be performed in* $O(\log^2 n)$ *parallel steps with* $n^2 p_{\mathbf{F}}(n) q_{\mathbf{F}}(n)/\log n$ *processors. Moreover,* A^{-1} *is still a Hankel-like matrix and its displacement generator can be computed in* $O(\log^2 n)$ *parallel steps with* $n^2 p_{\mathbf{F}}(n) q_{\mathbf{F}}(n)/\log n$ *processors.*
- *The computation of the matrix-vector product* $A\mathbf{x}$ *can be performed in* $O(\log n)$ *parallel steps with* $np_{\mathbf{F}}(n)$ *processors.*
- *The product of Hankel-like matrices is still a Hankel-like matrix, its displacement generator can be computed in* $O(\log n)$ *parallel steps with* $np_{\mathbf{F}}(n)$ *processors.*
- *The computation of the quotient and remainder of the division of two polynomials of degree at most n can be performed in* $O(\log n \log\log n)$ *parallel steps with* $np_{\mathbf{F}}(n)$ *processors.*

Let us describe the algorithm for the computation of $\gcd(u(x), v(x))$. Without loss of generality we assume that $m < n$. If $m = n$ then we consider the polynomials $u(x)$ and $\hat{v}(x) = v(x) \bmod u(x)$.

ALGORITHM 4.1. *Computing* gcd.

Input: natural numbers m and n, $m < n$, and the coefficients $u_0, \ldots, u_n$, $v_0, \ldots, v_m$ of two polynomials $u(x) = \sum_{i=0}^{n} u_i x^i$, $v(x) = \sum_{i=0}^{m} v_i x^i$.

Output: The coefficients of $\gcd(u(x), v(x)) = r(x)$.

Computation:

1. Compute rank $B(u, v) = k$. Set $l = n - k + 1$.
2. Solve the $k \times k$ system $C\mathbf{y} = \mathbf{b}$, $\mathbf{y} = (y_{k-1}, \ldots, y_0)^T$, where $C = (JB(u, v)J)_k$ and

$$\mathbf{b} = T(u_n, \ldots, u_l)(v_l, \ldots, v_{n-2k})^T - T(v_n, \ldots, v_l)(u_l, \ldots, u_{n-2k})^T.$$

3. Compute $\mathbf{w} = (B(u, 1)J)_{k+1}(y_0, \ldots, y_k)^T$, $y_k = 1$.
4. Compute $r(x)$ such that $u(x) = r(x)w(x)$.

Corollary 3.1 implies the correctness of the algorithm. Moreover, observe that the matrix $JB(u, v)J$ satisfies the condition of Proposition 4.1. Therefore all the computation can be performed in $O(\log^2 n)$ parallel steps with $n^2 p_{\mathbf{F}}(n) q_{\mathbf{F}}(n)/\log n$ processors.

In the view of Proposition 3.6 the above algorithm could be rewritten by replacing the matrix $JB(u, v)J$ with the Hankel matrix $H(u, v)$, the matrix C and the vector $\mathbf{b}$ at stage 2, with the matrix $(H(u, v))_k$ and the vector $(h_k, \ldots, h_{2k-1})^T$, respectively, obtaining the algorithm of [3]. However the entries of $H(u, v)$ may grow exponentially with n whereas the entries of $B(u, v)$ have moduli bounded by $\phi = 2n\mu\nu$, where $|u_i| \le \mu$, $|v_i| \le \nu$. Moreover,

if u and v have integer coefficients, then the matrix $B(u, v)$ has integer coefficients and all the computation can be kept within the integers.

It is possible to prove (see [5] and [13]) that if $u(x)$ and $v(x)$ have integer coefficients represented with d binary digits (bits), then the computation of Algorithm 4.1 can be performed with at most $O(n(d + \log n))$ bits. Moreover, if $u(x)$ and $v(x)$ have rational coefficients represented as pairs of d-digit integers, then $O(n^2(d + \log n)$ bits are sufficient to compute $\gcd(u, v)$ by using integer arithmetic.

ALGORITHM 4.2. *Computing the coefficients of the polynomials generated by the Euclidean scheme applied to $u(x)$ and $v(x)$ by means of the block triangular factorization of $JB(u, v)J$.*

Input: natural numbers m and n, and the coefficients $u_0, \ldots, u_n$, $v_0, \ldots, v_m$ of two polynomials $u(x) = \sum_{i=0}^{n} u_i x^i$, $v(x) = \sum_{i=0}^{m} v_i x^i$.

Output: The coefficients of the polynomials generated by the Euclidean scheme applied to $u(x)$ and $v(x)$.

Computation:

1. Compute $r(x) = \gcd(u(x), v(x))$ and replace $u(x)$, $v(x)$, n and m with $u(x)/r(x)$, $v(x)/r(x)$, $n - \deg r(x)$ and $m - \deg r(x)$, respectively.
2. If $n - m > n/2$ set $k = n - m$, otherwise compute rank $(JB(u, v)J)_{\lceil n/2 \rceil}$ and set $k = \text{rank } (JB(u, v)J)_{\lceil n/2 \rceil}$.
3. Let i be such that $k = m_i$ and factorize $JB(u, v)J$ as in (13). Compute

$$P_{m_i}^{-1}, \quad XP_{m_i}^{-1}, \quad S_{m_i} = W - XP_{m_i}^{-1}X^T$$

   and the coefficients of $r_i(x)$, $r_{i+1}(x)$, by using Proposition 3.4 and Remark 2.1.
4. Compute the coefficients of $u^{(i)}(x)$, $v^{(i)}(x)$ such that

$$H(u^{(i)}, v^{(i)}) = (H(u, v))_{m_i},$$

   that is, apply Proposition 2.1 to the matrix $(H(u, v))_{m_i}$, which is nonsingular for (10) and Remark 3.1.
5. Recursively apply Algorithm 4.2 to $JB(u^{(i)}, v^{(i)})J$, $JB(r_i, r_{i+1})J$, obtaining

$$JB(u^{(i)}, v^{(i)})J = L'D'L'^T, \qquad JB(r_i, r_{i+1})J = L''D''L''^T.$$

6. Set

$$L = \begin{pmatrix} I & O \\ XP_{m_i}^{-1} & I \end{pmatrix} \begin{pmatrix} T & O \\ O & I \end{pmatrix} \begin{pmatrix} L' & O \\ O & L'' \end{pmatrix}$$

   and obtain the coefficients of all the $r_i(x)$'s.
7. Recover the coefficients of the polynomial generated by the Euclidean scheme applied to $u(x)$ and $v(x)$, from the coefficients of the polynomial generated by the Euclidean scheme applied to $u(x)/r(x)$ and $v(x)/r(x)$.

The correctness of the algorithm follows from (13). Observe that stage 1 can be performed by Algorithm 4.1. Stage 2, which consists in the rank computation of a Hankel-like matrix, can be performed in $O(\log^2 n)$ steps with $n^2 p_{\mathbf{F}}(n) q_{\mathbf{F}}(n)/\log n$ processors. Concerning the computation at stage 3 we observe that, since P_{m_i}, $P_{m_i}^{-1}$, $XP_{m_i}^{-1}$ and S_{m_i} are Hankel-like matrices, it is sufficient to compute only their displacement generators for the cost of $O(\log^2 n)$ steps with $n^2 p_{\mathbf{F}}(n) q_{\mathbf{F}}(n)/\log n$ processors. Stage 4 is reduced to solving a Hankel system for the cost of $O(\log^2 n)$ steps with $n^2 p_{\mathbf{F}}(n) q_{\mathbf{F}}(n)/\log n$ processors. At stage 6, $O(\log n)$ steps

with $n^2/\log n$ processors are sufficient to compute the triangular factor L; the same complexity bound applies to stage 7.

Therefore each recursive step of the algorithm can be performed in $O(\log^2 n)$ parallel steps with $n^2 p_{\mathbf{F}}(n) q_{\mathbf{F}}(n)/\log n$ processors.

It is easy to check that $O(\log n)$ recursive steps are sufficient for the algorithm. In fact, from Remark 3.1 deg $r_i(x) = n - m_i$, and since deg $r_{i+1}(x) < n/2$ (i.e., $m_{i+1} > n/2$), then

$$\deg\ r_i(x) - \deg\ r_{i+1}(x) = m_{i+1} - m_i > n/2 - m_i.$$

Hence, by applying another recursive step to $JB(r_i, r_{i+1})J$ we obtain two diagonal blocks: the first of size $n/2 - m_i$ already factorized, the second of size $n/2$ (for simplicity we assume n even). Hence, the overall cost of Algorithm 4.2 is $O(\log^3 n)$ parallel steps with $n^2 p_{\mathbf{F}}(n) q_{\mathbf{F}}(n)/\log n$ processors.

With the same cost it is possible to compute also all the polynomials generated by the extended Extended Euclidean Scheme (EES) by solving at most n linear systems defined by the same Toeplitz-like matrix. In fact the polynomials $s_i(x)$ and $t_i(x)$ generated by EES,

$$\begin{array}{llll} s_0(x) = 1, & s_1(x) = 0, & s_i(x) = & s_{i-2}(x) - q_i(x)s_{i-1}(x), \\ t_0(x) = 0, & t_1(x) = 1, & t_i(x) = & t_{i-2}(x) - q_i(x)t_{i-1}(x), \end{array} \quad i = 2, \ldots, L,$$

satisfy the equation

$$s_i(x)u(x) + t_i(x)v(x) = r_i(x), \quad i = 1, 2, \ldots, L. \tag{14}$$

Rewriting (14) in matrix form we obtain

$$R\begin{pmatrix} \mathbf{s}_i \\ \mathbf{t}_i \end{pmatrix} = \mathbf{r}_i, \qquad i = 1, 2, \ldots, L,$$

where $\mathbf{s}_i, \mathbf{t}_i, \mathbf{r}_i$ are the vectors associated with the polynomials $s_i(x)$, $t_i(x)$, and $r_i(x)$, respectively, filled with zeros up to the dimension m, n, $m+n$, respectively, and R is the $(m+n) \times (m+n)$ resultant (Sylvester) matrix of $u(x)$ and $v(x)$, that is, $R = [T_1|T_2]$, where T_1 and T_2 are Toeplitz matrices of sizes $(m+n) \times m$ and $(m+n) \times n$, respectively, with the first rows $(u_n, 0, \ldots, 0)$ and $(v_m, 0, \ldots, 0)$ and the first columns $(u_n, u_{n-1}, \ldots, u_0, 0, \ldots, 0)^T$ and $(v_m, v_{m-1}, \ldots, v_0, \ldots, 0)^T$, respectively.

We may assume without loss of generality $\gcd(u(x), v(x)) = 1$, otherwise we apply Algorithm 4.1. Therefore R is nonsingular and

$$\begin{pmatrix} \mathbf{s_i} \\ \mathbf{t_i} \end{pmatrix} = R^{-1}\mathbf{r_i}, \qquad i = 1, \ldots, L.$$

The matrix R is Toeplitz-like and the following relation holds [5]:

$$R^{-1} = L_1U_1 + L_2U_2,$$

where L_1, U_1, L_2, and U_2 are suitable triangular Toeplitz matrices. Then the computation of the vectors $\mathbf{s}_i, \mathbf{t}_i$, $i = 1, \ldots, L$ can be performed in two stages:

1. compute the matrices L_1, U_1, L_2, U_2,
2. compute $(L_1U_1 + L_2U_2)\mathbf{r}_i$, $i = 1, \ldots, L$.

Stage 1 can be performed for a cost of $O(\log^2 n)$ steps with $n^2 p_{\mathbf{F}}(n) q_{\mathbf{F}}(n)/\log n$ processors [5]. Stage 2 consists in computing L simultaneous matrix-vector products with a Toeplitz-like matrix, so that the overall cost of the EES is dominated by the cost of computing the polynomials $r_i(x)$.

REFERENCES

[1] A. G. AKRITAS, *A new method for computing polynomial greatest common divisors and polynomial remainder sequences*, Numer. Math. 52 (1988), pp. 119–127.

[2] S. BARNETT, *Greatest common divisor of two polynomials*, Linear Algebra Appl., 3 (1970), pp. 7–9.

[3] D. BINI AND L. GEMIGNANI, *On the Euclidean scheme for polynomials having interlaced real zeros*, Proc. 2nd Ann. ACM Symp. on Parallel Algorithms and Architectures, 254-258, ACM Press, New York, 1990.

[4] D. BINI, L. GEMIGNANI, AND V. PAN, *Improved parallel computations with matrices and polynomials*, Proc. 18th ICALP, Lecture Notes in Comput. Sci., 510, Springer-Verlag, Berlin, 1991, pp. 520–531.

[5] D. BINI AND V. PAN, *Polynomial and Matrix Computations*, Vol. 1: Fundamental Algorithms. Birkhäuser, Boston, MA, 1994.

[6] ———, *Improved parallel polynomial division*, Proc. 33rd Ann. IEEE Symp. on Foundation of Comp. Science, 131-136, IEEE Computer Science Press, 1992; SIAM J. Comput., 22 (1993), pp. 617–626.

[7] A. BORODIN, J. VON ZUR GATHEN, AND J. HOPCROFT, *Fast parallel matrix and gcd computation*, Information and Control, 52 (1982), pp. 241–256.

[8] W. S. BROWN AND J. F. TRAUB, *On Euclid's algorithm and the theory of subresultants*, J. Assoc. Comput. Mach., 18 (1971), pp. 505–514.

[9] G. E. COLLINS, *Subresultants and reduced polynomial remainder sequences*, J. Assoc. Comput. Mach., 14 (1967), pp. 128–142.

[10] M. FIEDLER, *Hankel and Loewner matrices*, Linear Algebra Appl. 58 (1984), pp. 75–95.

[11] P. A. FUHRMANN AND N. B. DATTA, *On Bézout, Vandermonde matrices and the Lienard-Chipart Stability Criterion*, Linear Algebra Appl., 120 (1989), pp. 23.

[12] M. H. GUTKNECHT, *A completed theory of the unsymmetric Lanczos process and related algorithms, part* I, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 594–639.

[13] L. GEMIGNANI, *Solving Hankel systems over the integers,* Dipartimento di Matematica, Università di Parma, Quaderno 87, 1993, Parma, Italy; J. Symbolic Comput., to appear.

[14] W. B. GRAGG AND A. LINDQUIST, *On the partial realization problem*, Linear Algebra Appl., 50 (1983), pp. 277–319.

[15] A. S. HOUSEHOLDER, *The Numerical Treatment of a Single Nonlinear Equation*, McGraw-Hill, New York, 1970.

[16] U. HELMKE AND P. A. FUHRMANN, *Bézoutians*, Linear Algebra Appl., 124 (1989), pp. 1039–1097.

[17] G. HEINIG AND K. ROST, *Algebraic Methods for Toeplitz-like Matrices and Operators*, Oper. Theory 13, Birkhäuser, Boston, MA, 1984.

[18] A. E. KALTOFEN, *Processor-efficient parallel computation of polynomial greatest common divisor*, Preliminary Report, Dept. of Computer Sci., Rensselaer Polytechnic Institute, Troy, NY, 1989.

[19] P. LANCASTER AND M. TISMENETSKY, *The Theory of Matrices*, Academic Press, New York, 1985.

[20] V. PAN, *Parametrization of Newton's iteration for computations with structured matrices and applications*, Comput. Math. Appl., 24 (1992), pp. 61–75.

[21] B. PARLETT, *Reduction to tridiagonal form and minimal realizations*, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 105–124.

[22] D. PAL AND T. KAILATH, *Fast Triangular Factorization of Hankel and Related Matrices with Arbitrary Rank Profile*, Tech. Report, Information Systems Laboratory, Stanford University, Stanford, CA, 1990.

[23] H. K. WIMMER, *On the history of the Bézoutian and the resultant matrix*, Linear Algebra Appl., 128 (1990), pp. 27–34.

SIAM J. COMPUT.
Vol. 24, No. 1, pp. 78–100, February 1995

005

A GRAPH-THEORETIC GAME AND ITS APPLICATION TO THE k-SERVER PROBLEM*

NOGA ALON†, RICHARD M. KARP‡, DAVID PELEG§, AND DOUGLAS WEST¶

Abstract. This paper investigates a zero-sum game played on a weighted connected graph G between two players, the *tree player* and the *edge player*. At each play, the tree player chooses a spanning tree T and the edge player chooses an edge e. The payoff to the edge player is $cost(T, e)$, defined as follows: If e lies in the tree T then $cost(T, e) = 0$; if e does not lie in the tree then $cost(T, e) = cycle(T, e)/w(e)$, where $w(e)$ is the weight of edge e and $cycle(T, e)$ is the weight of the unique cycle formed when edge e is added to the tree T. The main result is that the value of the game on any n-vertex graph is bounded above by $\exp(O(\sqrt{\log n \log\log n}))$. It is conjectured that the value of the game is $O(\log n)$.

The game arises in connection with the k-server problem on a *road network*; i.e., a metric space that can be represented as a multigraph G in which each edge e represents a road of length $w(e)$. It is shown that, if the value of the game on G is $Val(G, w)$, then there is a randomized strategy that achieves a competitive ratio of $k(1 + Val(G, w))$ against any oblivious adversary. Thus, on any n-vertex road network, there is a randomized algorithm for the k-server problem that is $k \cdot \exp(O(\sqrt{\log n \log\log n}))$ competitive against oblivious adversaries.

At the heart of the analysis of the game is an algorithm that provides an approximate solution for the *simple network design problem*. Specifically, for any n-vertex weighted, connected multigraph, the algorithm constructs a spanning tree T such that the average, over all edges e, of $cost(T, e)$ is less than or equal to $\exp(O(\sqrt{\log n \log\log n}))$. This result has potential application to the design of communication networks. It also improves substantially known estimates concerning the existence of a sparse basis for the cycle space of a graph.

Key words. k servers, spanners, spanning trees, average stretch

AMS subject classifications. 05C05, 05C12, 05C85, 68R10, 90B80, 90D43

1. Introduction.

Let G be a connected multigraph, and let w be a function from the edge set of G into the positive reals; $w(e)$ is called the *weight* of edge e. Consider a two-person zero-sum game between a *tree player* and an *edge player*. At each play the tree player chooses a spanning tree T and, simultaneously, the edge player chooses an edge e. The payoff $cost(T, e)$ is defined as follows. For an edge e that does not lie in the tree T, let $cycle(T, e)$ denote the weight of the unique cycle formed when edge e is added to the tree T. Then

$$cost(T, e) = \begin{cases} 0 & \text{if } e \text{ lies in the tree } T, \\ cycle(T, e)/w(e) & \text{otherwise.} \end{cases}$$

Our main interest is in determining the value of this game for particular weighted multigraphs, and in determining an upper bound on the value in terms of the number of vertices.

We introduce some standard terminology. A mixed strategy for the tree player is a probability distribution p over the spanning trees of G, assigning to each spanning tree T a probability $p(T)$; similarly, a mixed strategy for the edge player is a probability distribution q over the edges, assigning to each edge a probability $q(e)$. The min-max theorem of game theory tells us that

$$\min_p \max_q \sum_T \sum_e p(T)q(e)cost(T, e) = \max_q \min_p \sum_T \sum_e p(T)q(e)cost(T, e).$$

*Received by the editors December 3, 1991; accepted for publication (in revised form) October 4, 1993.

†Tel Aviv University, Tel Aviv 69978, Israel. The research of this author was supported by a US-Israeli BSF Grant.

‡University of California at Berkeley and International Computer Science Institute, Berkeley, California 94720.

§The Weizmann Institute, Rehovot, Israel. The work of this author was supported by an Allon Fellowship, a Bantrell Fellowship, and a Haas Career Development Award.

¶University of Illinois. The research of this author was supported by Office of Naval Research Grant N00014-85K0570 and by NSA/MSP Grant MDA904-90-H-4011.

The common value of these two expressions is called the *value* of the game, and is denoted $Val(G, w)$. In the special *unweighted case* in which $w(e) = 1$ for all e, the value is denoted $Val(G)$.

Our main results with respect to the game are as follows:

- For every weighted n-vertex multigraph G, w,

$$Val(G, w) \leq \exp(O(\sqrt{\log n \log\log n})).$$

- There is an infinite sequence $\{G_n\}_{n\geq 1}$ of graphs such that G_n has n vertices and $Val(G_n) = \Omega(\log n)$.

We also provide a near-tight analysis of the game on several simple classes of graph topologies, including cycles, cycles with cross edges, grids and hypercubes.

At the heart of our analysis of the game on arbitrary multigraphs is an algorithm that provides an approximate solution for the *simple network design problem* of [JLR]. Specifically, for any n-vertex weighted, connected multigraph, our algorithm constructs a spanning tree T such that the average, over all edges e, of $cost(T, e)$ is less than or equal to $\exp(O(\sqrt{\log n \log\log n}))$. This algorithm provides us with a strategy for the tree player in the game. As a byproduct, our algorithm improves the main result of [SV] that deals with the choice of a sparse basis for the cycle space of a given graph. The authors of [SV] show that for every n-vertex graph there is a spanning tree so that the average length of a fundamental cycle is $O(\sqrt{n})$, whereas our result improves this estimate to $\exp(O(\sqrt{\log n \log\log n}))$.

A somewhat more general network design problem is the *optimum communication spanning tree problem* studied by Hu [Hu]. This problem extends the simple network design problem by introducing a *communication requirement matrix* on the vertices, and charging for each pair of vertices a cost proportional to their distance in the chosen spanning tree times their communication requirement. It can be shown that this seemingly more general problem is in fact reducible to the simple network design problem on multigraphs, so that any (approximate) solution for the latter problem can be translated into a solution for the former (with the same approximation ratio).

The game arises in connection with the k-server problem on a *road network*; i.e., a metric space that can be represented as a multigraph G in which each edge e represents a road of length $w(e)$. The class of these metric spaces seems to be one of the most natural ones for considering the k-server problem, as any real configuration of roads forms such a network. Note that any nontrivial road network has infinitely many points; a continuous circle, for example, can be represented by a road network with two vertices and two parallel edges joining them.

We show that if the value of the game on G is $Val(G, w)$, then there is a randomized strategy that achieves a competitive ratio of $k(1 + Val(G, w))$ against any oblivious adversary. Thus, on any n-vertex road network, there is a randomized algorithm for the k-server problem that is $k \cdot \exp(O(\sqrt{\log n \log\log n}))$ competitive against oblivious adversaries.

In addition, the spanning tree algorithm has potential application to the design of communication networks. Shortest-path trees are among the basic tools used in communication networks for various control tasks. Given a graph H, let $path(H, u, w)$ denote the weighted distance (i.e., the length of the shortest path) between u and w in H. A *shortest-path tree* for the graph H w.r.t. some vertex r is a tree T with the property that for every vertex v in H, its distance from r in T is the same as in H, i.e., $path(T, v, r) = path(H, v, r)$.

The obvious drawback of a shortest-path tree w.r.t. r is that while it yields optimal routes from r to any other node, the quality of the routes provided by the tree between other pairs of nodes may be poor. A natural measure for the quality of the path connecting nodes u, w in the tree T is its *stretch factor* (or dilation), defined as $path(T, u, w)/path(G, u, w)$. Classes of graphs for which there exist spanning trees with low worst-case stretch factor, termed *tree*

spanners, are studied in [Cai]. However, in general, looking for a tree with low worst-case stretch factor may be a hopeless task, since such a tree may not always exist. In particular, it is clear that there are graphs of diameter D for which any spanning tree incurs a worst-case stretch of $\Omega(D)$ for some pairs of nodes (the unit-weight cycle of two-D vertices is an example for such a graph). It may therefore be useful to look for trees that attempt to minimize the *average* stretch factor over all graph edges, or even over all pairs of vertices in the graph. This problem is referred to in [JLR] as the *simple network design problem*, and is shown therein to be NP hard. Our result provides a method for constructing such a tree, which in some cases may provide an attractive alternative to the standard shortest-path tree.

Let us comment that another viable alternative involves insisting on good bounds for the *worst-case* stretch, at the cost of allowing cycles in the spanning structure. It is known that for every graph there exist relatively sparse spanning subgraphs, termed *spanners*, guaranteeing this property [ADDJ], [PS], [PU]. However, the use of a *tree* as our spanning structure may sometimes be preferred due to its practical advantages, in terms of simplicity of the routing and control processes, lower total channel costs, and so on.

2. Basic examples. In this section we consider the game on several example graphs. The natural road networks that are not trees include cycles and grids. The solution of the game for the grid is difficult; asymptotics will appear in §6. Here we content ourselves with easier examples. A strategy for the edge player has value v if the minimum expected payoff for any tree against it is v, and a strategy for the tree player has value v if the maximum expected payoff against it for any edge is v; in other words, the value of a strategy is the payoff it ensures. The min-max theorem guarantees optimal strategies with the same value. Note that if the game has value v, then any choice given nonzero probability in the optimal strategy for one player must have expected payoff exactly v against the optimal strategy for the other player.

EXAMPLE 2.1. *The complete graph.* Let G be the unweighted n-vertex simple complete graph. If the edge player chooses uniformly among the edges, then selection of any tree will have probability $\frac{2}{n}$ of payoff 0 and probability $\frac{(n-2)}{n}$ of a positive payoff. The minimum positive payoff equals the length of the shortest cycle, which is 3. Therefore the expected payoff is at least $3 - \frac{6}{n}$. Equality holds only for the n-vertex stars, because other trees have fundamental cycles of lengths exceeding 3. If the tree player chooses uniformly among the n n-vertex stars, then any edge has probability $\frac{2}{n}$ of payoff 0 and probability $\frac{(n-2)}{n}$ of payoff 3, guaranteeing payoff at most $3 - \frac{6}{n}$. Hence uniform edge selection and uniform star selection are optimal strategies, and $Val(G) = 3 - \frac{6}{n}$. □

With arbitrary weights, the complete graph becomes rather complex. Therefore, let us consider simpler graphs and introduce weights.

EXAMPLE 2.2. *Cycles and multicycles.* If G is the n cycle, let T_i be the tree omitting edge i, and let w_i be the weight on edge i, with $W = \sum_i w_i$. If the tree player assigns probability $p_i = \frac{w_i}{W}$ to T_i, then every edge has expected payoff 1. If the edge player assigns probability $\frac{w_i}{W}$ to edge i, then every tree has expected payoff 1. Hence $Val(G, w) = 1$.

Now let G be the multigraph whose underlying simple graph is an n cycle, with G having n_i copies of edge i. Suppose that each copy of edge i has weight w_i, and let $W = \sum w_i$. Each tree consists of one copy of each of $n - 1$ edges. Let $\mathbf{T}_i$ denote the trees having no copy of edge i, and suppose these are chosen with equal probability totaling p_i. If $p_i = \frac{w_i}{W}$, as before, then the expected payoff for a copy of edge j is $1 + 2(1 - p_j)(1 - \frac{1}{n_j}) < 3$ (as in a multigraph, two parallel edges form a cycle of length 2). Hence $Val(G, w) < 3$. In fact, even for the unweighted case $Val(G)$ can be arbitrarily close to 3 when n and $\{n_i\}$ are appropriately chosen. In particular, if $w_i = 1$ and $n_i = m$ for all i, then when the edge player chooses edges uniformly the expected payoff of any tree is $1 + 2(1 - \frac{1}{n})(1 - \frac{1}{m})$, so $Val(G) \geq 3 - \frac{2}{n} - \frac{2}{m} + \frac{2}{nm}$. □

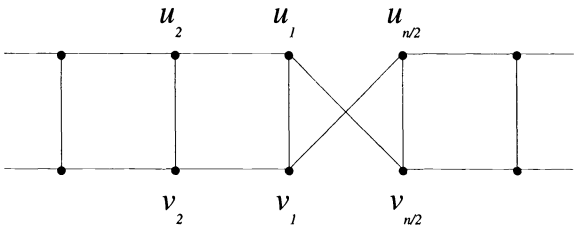


FIG. 1. *A portion of the twisted prism.*

In order to obtain higher values of the game for unweighted graphs, it is apparent that we need to have a substantial fraction of nontree edges, and that a large diameter will allow some of those edges to generate a large cost. (In any case, a small diameter leads to a small $Val(G)$.) The cycle meets the second criterion but fails the first, which suggests the following example.

EXAMPLE 2.3. *Cycles with diagonals.* Given n even, let G be the graph consisting of an n cycle C together with the chords joining antipodal points on the cycle. We prove $2-\frac{4}{3n} \le Val(G) \le 3-\frac{6}{n}$; we do not have an exact solution for this graph. The upper bound is achieved as follows. Let T be a tree consisting of a path of $\frac{n}{2}$ edges on C and the $\frac{n}{2}-1$ diagonals having one endpoint in the path. If the tree player plays the n rotations of T with equal probability, then the expected payoff of an edge of C is $0\cdot\frac{1}{2}+4\cdot(\frac{1}{2}-\frac{2}{n})+(\frac{n}{2}+1)\cdot(\frac{2}{n}) = 3-\frac{6}{n}$, and the expected payoff of a diagonal is $0\cdot(1-\frac{1}{n/2})+(n/2+1)\cdot\frac{1}{n/2} = 1+\frac{2}{n}$. This upper bound is not optimal, since in particular, the strategy of sticking to the edges of C (and choosing them uniformly) does not guarantee a payoff greater than 1 for the edge player.

The uniform edge strategy establishes the lower bound. A tree containing no diagonals has expected payoff $[n+(\frac{n}{2})(\frac{n}{2}+1)]\frac{2}{3n}$, which is quite large. (Indeed, such a tree is simply the cycle C except one of its edges. The cost of this edge with respect to the tree is n and the cost of each diagonal is $n/2+1$). For any other tree T, we claim there are at least two edges of C not in T that complete fundamental cycles of length at least $\frac{n}{2}+1$. Therefore, the expected cost of T is at least $[(\frac{n}{2}+1)2+4(\frac{n}{2}-1)]\frac{2}{3n} = 2-\frac{4}{3n}$ (since the cost of each nontree edge is clearly at least 4.) To prove the claim, view G as a Möbius band or twisted ladder (see Fig. 1); the cycle wraps around twice, and the diagonals become rungs of the ladder. Select a diagonal edge e in T. Label the vertices $u_1, \ldots, u_{n/2}$ counterclockwise from one end of e and $v_1, \ldots, v_{n/2}$ counterclockwise from the other end. Let F be the component of T containing e that is obtained from T by deleting either or both of $u_1v_{n/2}$ and $v_1u_{n/2}$ from T, if they are present. Let i [j] be the largest index such that $u_i \in F$ [$v_j \in F$]. Then $cost(T, u_iu_{i+1})$, $cost(T, v_jv_{j+1}) \ge \frac{n}{2}+1$, because the choice of i implies that the path in T between u_i and u_{i+1} must contain at least one of $\{u_k, v_k\}$ for each k in $\{i+1, \ldots, \frac{n}{2}, 1, \ldots, i\}$, and both of them for some k. This is similar for v_jv_{j+1}.

EXAMPLE 2.4. *Weighted cycle with diagonals.* Consider the graph G given in Example 2.3, but suppose the edges of C have weight 1 and the diagonals have weight $w > 1$. Assume that n is an even multiple of k. Let T_k be a tree obtained by taking $\frac{n}{2k}$ equally spaced diagonals, growing a path of length $\lceil\frac{k-1}{2}\rceil$ clockwise and $\lfloor\frac{k-1}{2}\rfloor$ counterclockwise from each endpoint of each diagonal taken, and adding cycle edges on one semicircle to connect these components. The cost of a nontree cycle edge is $2k+2w$, except for two edges with cost $\frac{n}{2}+w$. The total cost of a set of $k-1$ consecutive nontree diagonals is

$$2(k-1)+\frac{2}{w}\left[\binom{\lceil(k+1)/2\rceil}{2}+\binom{\lfloor(k+1)/2\rfloor}{2}\right] = 2(k-1)+\frac{1}{w}\lfloor\frac{k^2}{2}\rfloor.$$

If the rotations of T_k are selected with equal probability, then the expected cost of an edge of C is $\frac{1}{n}[(\frac{n}{2}+w)2+(2k+2w)(\frac{n}{2k}-1)] = 2+\frac{w}{k}-\frac{2k}{n}$. The expected cost of a diagonal (if k is even) is $2+\frac{k}{2w}-\frac{1}{k}$. If we choose $k \approx \sqrt{2}w$, then we obtain $Val(G, w) < 2.707\ldots$. □

All of our examples here have values bounded by constants. However, it is relatively easy to construct examples for which $Val(G) = \Omega(\log n)$. As is well known there are 4-regular graphs on n vertices with girth $c \log n$ (cf. [Bo], pp. 107–109). For any tree, the $n+1$ nontree edges each have cost at least $c \log n$. Hence against the uniform edge strategy every tree has expected cost at least $(\frac{c}{2}) \log n$. In §5, we will see that the N-vertex grid also has cost $\Theta(\log N)$.

3. An application: The *k*-server problem on an undirected network. Our game on graphs first suggested itself in connection with the *k-server problem*, which was introduced in [MaMcSl] and has been studied by many researchers. The problem is set in a metric space M and involves the use of k *servers* to process a sequence of *requests*. At any stage in the processing of the sequence of requests, each server is located at a point in M; the initial locations of the servers are stipulated as part of the problem. Each request is specified by a point $r \in M$. A request at r is processed by moving one of the servers from its present location to r; the cost of processing the request is the distance the server moves. A *deterministic on-line algorithm* is a deterministic rule for deciding which server to move in response to a request. The choice must be determined by the initial positions of the servers and the sequence of requests up to the present request. It is because the choice is not allowed to depend on knowledge of future requests that the algorithm is called on-line.

For any pair π, ρ, where π specifies the initial positions of the k servers and ρ specifies a finite sequence of requests, and for any deterministic on-line algorithm A, let $A(\pi, \rho)$ denote the cost that A incurs in processing the sequence of requests ρ, starting with the servers in the initial positions π. We may also associate with the pair π, ρ a real number $OPT(\pi, \rho)$ denoting the minimum possible cost of processing the request sequence ρ, starting with the servers in the k-tuple π of initial positions; $OPT(\pi, \rho)$ can be viewed as the cost that would be incurred by an *optimal off-line algorithm* that knew the entire request sequence, as well as the initial positions of the servers, in advance, and thus could determine the least expensive way of processing the entire request sequence.

Research on the k-server problem focuses on determining the factor in extra cost that a deterministic on-line algorithm must pay because of the handicap of making its decisions on-line, without knowing the future. This is formalized as follows. Let C be a positive constant. The deterministic on-line algorithm A is called *C-competitive* if there exists a positive constant a such that, for all pairs π, ρ,

$$A(\pi, \rho) \leq C \cdot OPT(\pi, \rho) + a.$$

In [MaMcSl] it is shown that for every positive integer k, every metric space M with at least $k+1$ distinct points, and every C less than k, there does not exist a C-competitive deterministic on-line algorithm. Thus, except in trivial cases, it is impossible to achieve a competitive factor less than k. On the other hand, in [FiRaRa] it is shown that for every positive integer k and every metric space M, there exists a C-competitive deterministic on-line algorithm for some C, where C depends on k but not on the metric space. Thus a bounded competitive factor is always achievable. Chrobak and Larmore [ChLa] consider a special type of metric space that might be called a *tree-like road network*. Such a network consists of a tree T in which each edge $\{u, v\}$ is an interval of length $w(u, v)$ between vertex u and vertex v. The metric space consists of the union of all these intervals. The distance between two points x and y is just the length of the unique simple path between x and y in the network. Chrobak and Larmore show constructively that for any tree-like road network, and any number of servers k, there is

a k-competitive deterministic on-line algorithm, matching the lower bound of k established in [MaMcSl].

It is also possible to consider randomized on-line algorithms, in which the server to move at each step is chosen by a random experiment which, of course, takes into account the initial positions of the servers, the sequence of requests up to the present one, and the choices made by the algorithm at previous steps. In the case of a randomized algorithm, the cost $A(\pi, \rho)$ is a random variable. In defining the competitive factor achieved by a randomized on-line algorithm, we view the request sequence as being specified by an *adversary* who is trying to foil the algorithm. We distinguish between two types of adversaries: the *oblivious adversary*, who specifies the entire sequence of requests in advance, and the *adaptive adversary*, who, in specifying the next request, can take into account the algorithm's responses to all previous requests. In this paper we restrict attention to oblivious adversaries. The randomized algorithm A is said to be *C-competitive* (against oblivious adversaries) if there exists a positive constant a such that for every k-tuple π of initial positions and every request sequence ρ,

$$E[A(\pi, \rho)] \leq C \cdot COST(\pi, \rho) + a,$$

where E denotes mathematical expectation. Several examples are known in which randomized algorithms can achieve a smaller competitive factor than deterministic ones [BBKTW], [BLS], [FKLMSY].

We consider the k-server problem on a class of metric spaces that generalize the tree-like road networks of [ChLa]. Such a space is specified by a multigraph G in which each edge e has a positive real weight $w(e)$. The edge $e = \{u, v\}$ may be viewed as an interval of length $w(e)$ between vertices u and v. The metric space $M(G, w)$ consists of the union of all these intervals; the distance between two points x and y is just the length of the shortest path between x and y; we shall sometimes refer to such a metric space as a *road network.*

The following theorem establishes a connection between our two-person game and the k-server problem on a road network.

THEOREM 3.1. *Let G be a multigraph and w a function assigning to each edge of G a positive real weight $w(e)$. Then for every k, there is a $k(1 + Val(G, w))$-competitive randomized on-line algorithm for the k-server problem on the road network $M(G, w)$.*

Proof. The algorithm is as follows.

- Using an optimal mixed strategy for the tree player in the game defined by G and w, select a spanning tree T in G.
- Along each edge e not in T, choose uniformly a random point $x(e)$, and place a "roadblock" at $x(e)$; more precisely, replace $e = \{u, v\}$ by two intervals $[u, x_1(e)$ and $[x_2(e), v]$, where $x_1(e)$ and $x_2(v)$ are new points, distinct from all points in the original metric space. The edge $[u, x_1(e)]$ is of the same length as the interval $[u, x(e)]$, and the interval $[x_2(e), v]$ is of the same length as the interval $[x(e), v]$. This transformation converts the original road network to a tree-like road network with the same set of points but a different distance function.
- Process the request sequence by executing the (deterministic) Chrobak–Larmore algorithm on this derived tree-like road network.

Let the random variable $A(\pi, \rho)$ denote the cost that the above randomized algorithm incurs in processing the request sequence π, starting from the initial server positions ρ. Let $OPT(\pi, \rho)$ denote the optimal off-line cost of processing ρ, starting from server positions π, on the road network defined by multigraph G and weight function w. Let the random variable $OPT'(\pi, \rho)$ denote the optimal off-line cost of processing ρ, starting from the server positions π, in the tree-like road network constructed by the algorithm. By the k-competitiveness of the Chrobak–Larmore algorithm, $A(\pi, \rho) \leq k \cdot OPT'(\pi, \rho) + a$. To complete the proof, we

shall show that

$$E[OPT'(\pi, \rho)] \leq (1 + Val(G, w))OPT(\pi, \rho). \tag{1}$$

Suppose that the tree player has selected a spanning tree T of G and then for each nontree edge e has placed a roadblock at a random point $x(e)$. Consider a particular sequence of moves on the original road network that processes the sequence of requests ρ starting from the initial k tuple of server positions π at cost $OPT(\pi, \rho)$. Let $\rho = \rho_1\rho_2 \ldots \rho_t$. For $i = 1, 2, \ldots, t$, let $x(i)$ be the index of the server that processes request i. Then it is possible to satisfy ρ on the derived tree-like road network by having server $x(i)$ process request ρ_i, for each i. The server may not be able to follow the same path it would have followed on the original network, because of the presence of roadblocks. However, the server can simulate the path it would have followed on the original road network, except that whenever it encounters a roadblock $x(e)$, it traverses the unique path in the tree-like road network between $x_1(e)$ and $x_2(e)$ in order to get to the other side of the roadblock. The cost of each such detour is $cycle(T, e)$, the weight of the unique cycle formed by the edge e with the tree T.

We now compute the expected cost of all the detours, given that the tree player uses an optimal mixed strategy. Let $p(T)$ be the probability that the tree player chooses spanning tree T. Let $d(e)$ be the distance that servers travel along edge e in a particular sequence of moves achieving cost $OPT(\pi, \rho)$ in the original network; then

$$\sum_e d(e) = OPT(\pi, \rho).$$

If $x(e)$ is a randomly chosen point along edge e, then the expected number of times that servers cross the point $x(e)$ is $\frac{d(e)}{w(e)}$. Each crossing of the point $x(e)$ requires a detour of cost 0 if e lies in T and $cycle(T, e)$ if e does not lie in T. Thus, the expected cost of detours associated with the edge e is exactly $d(e)cost(T, e)$, and the expected total cost of detours is

$$\sum_T \sum_e p(T)d(e)cost(T, e) = OPT(\pi, \rho) \sum_T \sum_e p(T)q(e)cost(T, e),$$

where

$$q(e) = \frac{d(e)}{OPT(\pi, \rho)}.$$

But since the sequence of numbers $\{q(e)\}$ constitutes a probability distribution, it follows that the expected sum of detours is bounded above by $OPT(\pi, \rho)Val(G, w)$, and hence (1) follows. □

COROLLARY 3.2. *There is a $2k$-competitive randomized algorithm on the circle.*

Proof. This follows immediately from Example 2.2 (or even from its special case corresponding to a cycle of two vertices). □

Note that as shown in [FRRS] there is a *deterministic* $O(k^3)$-competitive algorithm for the circle.

4. An optimization problem related to the tree game. In this section we introduce a natural optimization problem and show that it is closely related to our graph-theoretic game. Let G be a connected multigraph with edge multiset E and let w be a function that assigns to each edge e in G a positive weight $w(e)$. Assume that the edge weights are normalized so that the lightest edge has weight 1. For any spanning tree T of G, and any edge e of G, let $path(T, e)$ denote the weight of the path in T between the endpoints of e. Define

$$cost^*(T, e) = path(T, e)/w(e) .$$

Also, for every subset of edges E', denote

$$cost^*(T, E') = \sum_{e \in E'} cost^*(T, e).$$

Let $S(G, w, T) = cost^*(T, E)/|E|$. Let $S_{opt}(G, w) = \min_T S(G, w, T)$, where T ranges over all spanning trees of G. In the unweighted case, where $w(e) = 1$ for all e, we abbreviate $S(G, w, T)$ by $S(G, T)$ and $S_{opt}(G, w)$ by $S_{opt}(G)$. We shall show that the problem of finding a spanning tree that minimizes $S(G, w, T)$ is closely related to the problem of finding optimal strategies for the tree player and edge player in a variant of our tree game. In this new game, the payoff to the edge player when the edge player chooses edge e and the tree player chooses spanning tree T is $cost^*(T, e)$. Let the value of this game be denoted $Val^*(G, w)$. In the unweighted case, we abbreviate $Val^*(G, w)$ by $Val^*(G)$. Then, since $|cost(T, e) - cost^*(T, e)| \leq 1$ for every edge e, $|Val(G, w) - Val^*(G, w)| \leq 1$. Thus, our new game is very closely related to the original one.

Let G, w be a weighted multigraph. The operation of *replication* of an edge e replaces the edge e by one or more parallel edges of weight $w(e)$ between the endpoints of e. Any weighted multigraph created from G, w by a sequence of such operations is said to be *obtained from G, w by replication.*

THEOREM 4.1. *$Val^*(G, w) = \sup_{G', w'} S_{opt}(G', w')$, where G', w' ranges over all weighted multigraphs obtained from G, w by replication.*

Proof. Recall that

$$Val^*(G, w) = \max_q \min_p \sum_T \sum_e p(T)q(e)cost^*(T, e),$$

where q ranges over probability distributions for the edge player, and p over probability distributions for the tree player. For each fixed probability distribution q, $\sum_T \sum_e p(T)q(e)$ $cost^*(T, e)$ is minimized by a probability distribution p concentrated on one tree; this is an instance of the general observation that if one of the players in a zero-sum two-person game announces his mixed strategy, then the other player can play optimally by choosing a pure strategy. It follows that

$$Val^*(G, w) = \max_q \min_T \sum_e q(e)cost^*(T, e).$$

Now consider any weighted multigraph G', w' obtained from G, w by replication. Let $d(e)$ be the number of copies of edge e, and let $q(e)$ be $d(e)/(\sum_e d(e))$. Then q is a probability distribution, and

$$S_{opt}(G', w') = \min_T \sum_e q(e)cost^*(T, e).$$

It follows that $S_{opt}(G', w') \leq Val^*(G, w)$. Conversely, let q be the probability distribution for the edge player that maximizes $\min_T \sum_e q(e)cost^*(T, e)$. For any (large) integer M, let G^M, w^M be obtained from G, w by making $1 + \lfloor Mq(e) \rfloor$ copies of each edge e. Then clearly, as M tends to infinity, $S_{opt}(G^M, w^M)$ converges to $Val^*(G, w)$. The conclusion of the theorem follows. □

An *edge-transitive* graph is a graph G such that for any edges e, e' in G there is an automorphism σ in the automorphism group $\Gamma(G)$ such that σ takes the endpoints of e to the endpoints of e'. In the case of an unweighted edge-transitive graph, we prove that the optimization problem solves the game.

THEOREM 4.2. *If G is edge-transitive, then $Val^*(G) = S_{opt}(G)$.*

Proof. If the edge player plays each edge with equal probability, the expected payoff is at least $S_{opt}(G)$ for any tree. Let T be a tree with $S(G, T) = S_{opt}(G)$ and let $\mathbf{T}$ be the collection of images of T under the elements of $\Gamma(G)$. For each $T' \in \mathbf{T}$, let the probability of playing $\sigma(T)$ be $\frac{1}{|\Gamma(G)|}$ times the number of automorphisms σ such that $\sigma(T) = T'$.

We claim that the expected payoff for this tree strategy is $S_{opt}(G)$ for every edge, which completes the proof. Note that $cost^*(T, e) = cost^*(\sigma(T), \sigma(e))$. Also, Lagrange's theorem states that if a group ($\Gamma(G)$) acts on a set S (= $E(G)$), then the number of group elements taking $e \in S$ to any member of its orbit (including to itself) is the same. Hence the expected payoff for edge e is

$$\begin{aligned}\tfrac{1}{|\Gamma|}\textstyle\sum_\sigma cost^*(\sigma(T), e) &= \frac{1}{|\Gamma|}\sum_\sigma cost^*(T, \sigma^{-1}(e)) = \frac{1}{|\Gamma|}\frac{|\Gamma|}{|E|}\sum_{e'} cost^*(T, e') \\ &= S_{opt}(G). \qquad \square\end{aligned}$$

5. Upper and lower bounds for the tree game.

5.1. A lower bound. We begin with a lower bound on $Val^*(G)$.

THEOREM 5.1. *There exists a positive constant c such that for all n, there exists an n-vertex graph G_n such that $Val^*(G_n) \geq c \ln n$.*

Proof. The following is a known result in extremal graph theory (cf. [Bo], pp. 107–109): There exists a positive constant a such that for all n there exists an n-vertex graph G_n with $2n$ edges such that every cycle in G_n is of length at least $a \ln n$. Let T be any spanning tree in G. Then, for any nontree edge e, $cost^*(T, e) \geq a \ln n - 1$. Since more than half the edges are nontree edges, it follows that for every T, $\frac{1}{|E|}\sum_e cost^*(T, e) \geq \frac{1}{2}(a \ln n - 1)$. Thus $S_{opt}(G) \geq \frac{1}{2}(a \ln n - 1)$, and it follows from Theorem 4.1 that $Val^*(G) \geq \frac{1}{2}(a \ln n - 1)$. $\square$

We next give a preliminary result showing that for bounding $Val^*(G)$ from above, it is sufficient to consider multigraphs with at most $n(n+1)$ edges, counting multiplicities. (Note that in the context of the k-server problem we are really interested only in graphs. However, the construction technique employed in our proof makes it necessary to prove the result in the more general setting of multigraphs.)

LEMMA 5.2. *For every n-vertex multigraph G, w, there exists a multigraph G', w' on the same set of vertices and at most $n(n+1)$ edges such that $S_{opt}(G, w) \leq 2 \cdot S_{opt}(G', w')$.*

Proof. Let E be the edge multiset of G, and let E^{set} be the (maximal) set of distinct edges in E, where two edges are considered distinct if they don't have the same pair of endpoints. (That is, E^{set} contains a single representative edge uv for every pair of vertices u and v that are adjacent in G.) For each edge $e \in E^{set}$, let $d(e)$ be the number of copies of e in E, and let $w'(e)$ be the lowest weight of a copy of e in G, w. Then the cardinality of E (i.e., the total number of edges in G counting repetitions) is

$$|E| = \sum_{e \in E^{set}} d(e).$$

Consider a new multigraph G' with the same set of distinct edges, but with each edge e occurring $r(e)$ times instead of $d(e)$ times, where

$$r(e) = 1 + \left\lfloor \frac{d(e)|E^{set}|}{|E|} \right\rfloor . \tag{2}$$

Then the cardinality of the edge multiset E' of G' satisfies

$$|E'| = \sum_{e \in E^{set}} r(e) \leq |E^{set}| + \frac{|E^{set}|}{|E|}\sum_{e \in E^{set}} d(e) = 2|E^{set}| . \tag{3}$$

Since E^{set} contains at most one edge per pair of endpoints, including self-loops, it follows that G' has at most $n(n+1)$ edges, as required.

It remains to bound $S_{opt}(G', w')$. Combining (2) and (3) we get

$$r(e) \geq \frac{d(e)|E^{set}|}{|E|} \geq \frac{d(e)|E'|}{2|E|} . \tag{4}$$

The multigraph G' has a spanning tree T such that

$$S_{opt}(G', w') = \frac{1}{|E'|} \sum_{e \in E^{set}} r(e) cost^*(T, e, w') , \tag{5}$$

with the notation $cost^*(T, e, u)$ denoting $cost^*$ defined with respect to a weight function u. Using (4) and (5) and the choice of w', we find that

$$S_{opt}(G', w') \geq \frac{1}{2|E|} \sum_{e \in E^{set}} d(e) cost^*(T, e, w) .$$

But since T is a spanning tree of G as well as G', this last expression is at least $S_{opt}(G, w)/2$, so $S_{opt}(G, w) \leq 2S_{opt}(G', w')$. □

5.2. An upper bound for unweighted graphs: outline. Before we state and prove our upper bound for $S_{opt}(G, w)$ on general (weighted) graphs, it is instructive to sketch the construction in the simpler setting of an unweighted graph.

Our construction is based on the concepts of clusters and partitions. A *cluster* is a subset of the vertices whose induced subgraph is connected. A *partition* of a given graph $G = (V, E)$ is a collection of disjoint clusters whose union is V. The basic procedure used in the construction is a variant of the clustering algorithm of [Aw]. The construction involves a parameter x depending on n. This parameter is required to satisfy $x(n) > 1$ and be monotone increasing in n (with $x(n) \to \infty$), but its precise definition is left to be specified later. The output of the procedure is a partition of the given graph into clusters with low radii (specifically, radii at most $y(n) = O(x(n) \ln n)$), with the additional property that "most" of the graph edges are internal to clusters, and only a fraction of $1/x(n)$ of the edges connect endpoints in different clusters. We refer to edges of these two types as "internal" and "intercluster" edges, respectively.

A spanning tree can be built on the basis of such a partition as follows. First, construct a shortest-path spanning tree T_C (as defined earlier) for every cluster C in the partition. Note that since the partition is composed of disjoint clusters, these spanning trees form a spanning forest in the graph. Now connect the forest into a single tree by selecting a suitable tree of intercluster edges.

This last step can be carried out recursively. Given the partition, create an auxiliary *multigraph* $\tilde{G}$ by collapsing each cluster C into a single vertex v_C, and including an edge between two such vertices of $\tilde{G}$ for *each* original edge connecting these clusters. (Here is why it is useful to handle multigraphs in this algorithm, rather than simple graphs.) Next, apply the same procedure recursively to $\tilde{G}$, and obtain a tree $\tilde{T}$. The final tree T will consist of the union of $\tilde{T}$ and the trees T_C constructed for each cluster C.

Observe that internal edges will typically enjoy a lower cost than intercluster edges. This is because the path connecting the endpoints of an edge uw internal to a cluster C in the final tree T is the path connecting u and w on the tree T_C, and the partitioning algorithm guarantees that C has a low radius. In contrast, for an intercluster edge u_1u_2, where $u_1 \in C_1$ and $u_2 \in C_2$, the path connecting u_1 and u_2 in T is not simply the path connecting v_{C_1} and v_{C_2} in the tree $\tilde{T}$. Rather, this path is expanded when retracting from $\tilde{G}$ to G by replacing each vertex v_C on

it with an appropriate path segment on the internal tree T_C. This heavier cost for intercluster edges is compensated for by the fact that these edges are only a $1/x(n)$ fraction of the entire edge set, and therefore their contribution to the average cost is controllable.

The cost analysis is carried along the following lines. Let $f(n, m)$ be the maximum of $S(G, T)$ over all connected multigraphs G with up to n vertices and m edges, where T is the spanning tree of G constructed by the above algorithm. Then

$$f(n, m) \leq 2y(n) + \frac{1}{x(n)} \cdot f(n, m/x(n)) \cdot 5y(n) .$$

This inequality can be explained as follows. The first term in the right-hand side of the inequality denotes the contribution of the internal edges. This term is bounded above by $2y(n)$ since each cluster is of radius at most $y(n)$. The second term denotes the contribution of the intercluster edges. In this term, the factor $1/x(n)$ is an upper bound on the fraction of intercluster edges. The number of edges in $\tilde{G}$ is at most $m/x(n)$, thus the expected cost of an edge in $\tilde{G}$ is at most $f(n, m/x(n))$. Now consider an intercluster edge $e = uv$, and let $\tilde{\gamma}$ be the path connecting its endpoints in $\tilde{G}$, namely, $\tilde{\gamma} = (e_1, \ldots, e_q)$, where each $e_i = u_i v_i$ is an intercluster edge connecting the vertices $v_{C_{i-1}}$ and v_{C_i} in $\tilde{G}$, for $1 \leq i \leq q$, and $u \in C_0$ and $v \in C_q$. The path $\tilde{\gamma}$ can be expanded in G to a path γ connecting u and v, by taking the following steps:

1. inserting between e_i and e_{i+1} a subpath of the tree spanning C_i, connecting v_i and v_{i+1} (for $1 \leq i \leq q-1$),
2. inserting a subpath of the tree spanning C_0, connecting u and v_1, and
3. inserting a subpath of the tree spanning C_q, connecting v_q and v.

Each of these $q+1$ subpaths is of length at most $2y(n)$, namely, twice the radius of the corresponding cluster. Therefore the total length of the resulting path γ, for a given path $\tilde{\gamma}$ of length q, is at most $q + 2(q+1)y(n)$. Thus each connecting path in $\tilde{G}$ expands by a factor of at most $(q + 2(q+1)y(n))/q \leq 5y(n)$ in G. This accounts for the factor $f(n, m/x(n)) \cdot 5y(n)$ in the second term.

Finally, using the above inequality and choosing $x(n)$ optimally as $\exp(\sqrt{\ln n \ln\ln n})$, we find that $f(n) \leq \exp(O(\sqrt{\ln n \ln\ln n}))$.

5.3. An upper bound for weighted graphs: outline. For understanding the weighted case, it is convenient to think of the above algorithm as an iterative, rather than recursive one. From this point of view, think of the main clustering procedure as a "machine," to which the graph is "fed" for a number of iterations. In each iteration $j \geq 1$, the procedure constructs a partition for the current graph G_j (where G_1 is the original graph G), and then contracts the clusters into single vertices, thus creating the graph G_{j+1} to be processed in the next iteration. Each such iteration also reduces the number of "uncovered" edges by a factor of x, until all edges are "covered." (An edge is *covered* if it is internal to some cluster we have already constructed.)

Given this view of the partitioning process, one can analyze the radii of the clusters constructed at each iteration $j \geq 1$ (henceforth called "j clusters"). For that purpose, it is instructive to review the partitioning process in more detail. The partitioning procedure creates the clusters sequentially. Each cluster C is "grown" by starting at a single vertex, and successively merging it with layers of neighbouring vertices. This merging process is stopped once the number of outgoing edges of the current cluster C (namely, edges with one endpoint in C and one outside C) is at most a $1/x(n)$ fraction of the number of internal edges (namely, edges with both endpoints in C). As will be shown later on (in the formal proof for the weighted case), this merging process is bound to halt after adding no more than, say, $y(n)/3$ layers. Hence each j cluster has radius at most $y(n)/3$ *in the current graph* G_j. However,

in the original graph G, such j clusters have radius $r_j \leq y^{j+1}(n)$. This can be easily argued by noting that when a j cluster is constructed, each merged layer increases the radius by an additive factor of up to

$$1 + 2r_{j-1}, \tag{6}$$

where the 1 is contributed by the added edge, and the $2r_{j-1}$ by the diameter of the $(j-1)$ cluster corresponding to the endpoint of that edge in G_j. The bound on r_j follows inductively since at most $y(n)/3$ layers are merged.

Let us next outline the modifications needed for handling the weighted case. The main problem that needs to be overcome is that in this case, all edges cannot be treated alike, since when growing a cluster, a single layer merging step will increase the radius of the resulting cluster by the weight of the heaviest merged edge, rather than by just one, thus the radii of constructed clusters cannot be controlled.

The algorithm thus needs to be modified as follows. It is necessary to break the set of edges E into classes E_i, $i \geq 1$, according to weights, with E_i containing all edges whose weight is in the range $[y^{i-1}, y^i)$ for an appropriately chosen parameter $y = y(n, x)$ (with x a parameter to be determined in the same spirit as in the unweighted case). Intuitively, we would like to handle the lighter edges first. (We may, and will, assume that the minimum weight of an edge is 1.)

We now feed the classes E_i to the "machine" in a pipelined fashion, with overlaps. Namely, in iteration 1 only the edges of E_1 are considered, in the next iteration E_2 is added, and so on. In general, the class E_i is taken into consideration for the first time in the ith iteration, and is processed for the next $\rho = O(\ln n / \ln x)$ iterations, each reducing the number of unsatisfied edges in it by a factor of x, until the entire set E_i is exhausted.

A crucial point that must be explained at this point is the role of the parameter y in the construction. In the weighted case, this parameter has two different functions. The first is similar to the one it had in the unweighted case, i.e., it is (more or less) the radius increase bound for constructed clusters. That is, clusters constructed for the graph G_j in the jth iteration will have at most radius $y/3$ in G_j. The second function of the parameter y is governing the weight range of the edge classes.

The combination of these two functions implies that in the construction of new clusters during a given iteration j, there is a balance between the contributions to the radius made by previously constructed clusters and by new edges. This is what guarantees that cluster radii are properly bounded *in the original graph* G as well. Specifically, the radius of a j cluster (constructed in iteration j) is bounded by $r_j \leq y^{j+1}$. Formally, this can again be deduced inductively, noting that when a j cluster is constructed, each merged layer increases the radius by up to

$$y^j + 2r_{j-1} \leq 3y^j, \tag{7}$$

where (in analogy with (6)) the y^j is contributed by the added edge and the $2r_{j-1}$ by the diameter of the $(j-1)$ cluster corresponding to the endpoint of that edge in G_j, and $r_{j-1} \leq y^j$ follows by the inductive hypothesis. This, combined with the fact that at most $y/3$ layers are added, yields the desired bound of $r_j \leq y^{j+1}$ by induction.

5.4. The upper bound for weighted graphs: formal proof. We are now ready to formally state and prove the upper bound on S_{opt}.

THEOREM 5.3. *There exists a constant c such that for n sufficiently large, every n-vertex connected multigraph G and every weight function w satisfy* $S_{opt}(G, w) \leq \exp(c\sqrt{\log n \log\log n})$.

By Lemma 5.2, it suffices to prove the theorem for every n-vertex multigraph having at most $n(n+1)$ edges in its edge multiset.

We shall use an iterative construction to obtain a suitable spanning tree of G. Again, the construction involves a parameter $1 \leq x \leq n$ depending on n, to be fixed later. Define

$$\rho = \lceil \frac{3 \ln n}{\ln x} \rceil, \quad \mu = 9\rho \ln n, \quad y = x\mu.$$

Break the edge multiset E into subclasses E_i, for $i \geq 1$, according to weights, defining

$$E_i = \{e \mid w(e) \in [y^{i-1}, y^i)\}.$$

(Recall that the edge weights are normalized to be greater than or equal to 1.)

As described above, the algorithm proceeds in iterations. In each iteration j we compute some clusters in the graph G_j, and mark some of the edges as "covered." We then contract the clusters into single nodes, thus preparing the multigraph G_{j+1} for the next iteration.

More precisely, let E_i^j, for $i, j \geq 1$, denote the multiset of edges from E_i that are still uncovered at the beginning of iteration j. The purpose of the construction at iteration j is to partition the vertex set into disjoint clusters such that

- each cluster has a spanning tree of radius at most y^{j+1} in G;
- in every nonempty edge class E_i, $1 \leq i \leq j$, the fraction of intercluster edges is at most $1/x$; i.e., $|E_i^{j+1}| \leq |E_i^j|/x$.

Note that this definition for the purpose of an iteration, and particularly, restricting the second requirement to classes E_i for $i \leq j$, implies that iteration j need only consider (and process) edges from the edge multisets E_i for $i \leq j$, i.e., edges of weight less than y^j, and may ignore heavier edges. Avoiding heavier edges is crucial for guaranteeing the radius bound in the first requirement, as discussed earlier.

We next present the procedure used for forming the partition in iteration j. This procedure is a modified version of the clustering algorithm of [Aw]. The partition is constructed in a "greedy" fashion by a sequence of stages, each stage building a single cluster. After each stage, all the vertices of the graph that were included in the constructed cluster are eliminated from the graph.

Consider the beginning of a stage and let K be a connected component of the subgraph induced by the remaining vertices (i.e., the ones not yet included in any of the previously built clusters). Choose arbitrarily a "root vertex" u in K. Stratify the vertices and edges of K into layers according to their (unweighted) distance from u as follows. For each integer $\ell \geq 0$, let $V(\ell)$ be the set of vertices at (unweighted) distance ℓ from u in K (where the *unweighted distance* between two vertices is defined as the minimal number of edges in a path connecting them). Also let $E_i^j(\ell)$ be the set of edges of E_i^j that join a vertex in $V(\ell)$ with a vertex in $V(\ell) \cup V(\ell-1)$. Let ℓ^* be the least ℓ such that

$$(*) \qquad \forall 1 \leq i \leq j, \quad |E_i^j(\ell+1)| \leq \frac{1}{x}|E_i^j(1) \cup E_i^j(2) \cup \cdots \cup E_i^j(\ell)|.$$

Note that such ℓ always exists. The resulting cluster is the vertex set $V(1) \cup V(2) \cup \cdots \cup V(\ell^*)$. The vertices of this cluster are now eliminated from the graph, and the next stage is started. This process of cluster generation continues until the graph is exhausted (i.e., all vertices are assigned to clusters).

Let $x(n)$ be a function of n that will be specified later. The spanning tree T in an n-vertex multigraph G is constructed as follows:

- Set $j = 1$ and $G_j = G$.
- Set $x = x(n)$, ρ, μ, y, and the edge classes E_i, as defined above.
- While $\bigcup_i E_i \neq \emptyset$ do the following:
  1. Partition the vertex set of G_j into clusters as described above.
  2. Construct a shortest-path spanning tree T_C in each cluster C of G_j.
  3. Add each edge e of each of the constructed trees to the output tree T.
  4. Construct the next multigraph G_{j+1} by contracting each cluster C into a single vertex v_C, discarding internal edges from $\bigcup_i E_i$, and replacing each intercluster edge u_1u_2, for $u_1 \in C_1$ and $u_2 \in C_2$, by a new edge connecting the corresponding contracted vertices v_{C_1} and v_{C_2}.
  5. Set $j = j + 1$.

In order to analyze the output of our algorithm, we first bound the number of layers added to a cluster during the clustering process in some iteration j. This growth is bounded by showing that in iteration j, the only sets E_i^j considered by the algorithm (i.e., the only nonempty ones) are those satisfying $j - \rho \leq i \leq j$.

LEMMA 5.4. $|E_i^j| \leq |E_i|/x^{j-i}$ *for every* $1 \leq i \leq j$.

Proof. The claim follows from the fact that $|E_i^{j+1}| \leq |E_i^j|/x$ for every $1 \leq i \leq j$. To see this, note that by definition of the cluster construction process, each time a cluster is constructed in a connected component K, the number of edges of E_i^j connecting that cluster to the vertices of K not selected for inclusion is at most $\frac{1}{x}$ times the number of edges of E_i^j included in the cluster. Thus, in the entire graph the number of edges of E_i^j joining vertices in different clusters (constituting E_i^{j+1}) is at most $\frac{1}{x}$ times $|E_i^j|$. □

The above lemma enables us to bound the (weighted) radius of clusters generated during the execution.

LEMMA 5.5. *In iteration* j, *the radius of each cluster in* G *is bounded above by* y^{j+1}.

Proof. Let B_j denote the number of layers ℓ^* merged into any cluster in iteration j. We first prove that B_j is bounded above by

$$B_j \leq y/3 = 3\rho x \ln n. \tag{8}$$

To see this, note that by Lemma 5.4, $|E_i^{i+\rho}| = 0$ for every $i \geq 1$, which implies that the augmentation rule $(*)$ governing the addition of layers to the cluster need only consider edges from classes E_i^j for $j - \rho + 1 \leq i \leq j$. Each additional layer ℓ implies that some set $E_i^j(\ell)$ fails to satisfy this condition. Namely, for all ℓ such that $1 < \ell < \ell^*$, at least one i such that $j - \rho + 1 \leq i \leq j$ satisfies

$$|E_i^j(\ell)| > \frac{1}{x}|E_i^j(1) \cup E_i^j(2) \cup \cdots \cup E_i^j(\ell - 1)|.$$

Thus

$$|E_i^j(1) \cup E_i^j(2) \cup \cdots \cup E_i^j(\ell)| > \left(1 + \frac{1}{x}\right)|E_i^j(1) \cup E_i^j(2) \cup \cdots \cup E_i^j(\ell - 1)|.$$

Since there are at most ρ possible values of i, by the pigeonhole principle it follows that for ℓ^*, there exists at least one such i that caused the addition of a layer for at least $\lceil \ell^*/\rho \rceil$ times. This i satisfies

$$|E_i^j(1) \cup E_i^j(2) \cup \cdots \cup E_i^j(\ell^*)| \geq \left(1 + \frac{1}{x}\right)^{\lceil \ell^*/\rho \rceil}.$$

Since the number of edges in the multigraph is at most $n(n+1)$, it follows that, for n sufficiently large, $\ell^*/\rho \leq 3x \ln n$, so (8) is proved.

We now prove the main claim of the lemma, i.e., the bound on the radius of a cluster created in the jth iteration by induction on j. In order to analyze all iterations together (including also iteration 1), hypothesize iteration 0 as the one that yielded the initial graph G, with each vertex representing a cluster of radius 0 (thus trivially satisfying the inductive claim). Now for $j \geq 1$, suppose the claim holds up to $j-1$, and consider the jth iteration. At the beginning of the iteration, every vertex of the multigraph represents a cluster of radius up to y^j, by the inductive hypothesis. Also, every edge considered in iteration j is of length up to y^j. Hence in constructing a cluster, each additional layer contributes up to $3y^j$ to the radius. By inequality (8) above, the final radius is at most $3y^j \cdot y/3 = y^{j+1}$, as required. □

Using the above result we can bound the costs incurred by the edges, obtaining the following.

LEMMA 5.6. $cost^*(T, E) \leq 4x^2\mu^{\rho+1}|E|$.

Proof. Let H_i^j denote the set of edges from E_i that were covered during iteration j, i.e.,

$$H_i^j = E_i^j \setminus E_i^{j+1}.$$

We first claim that for every edge $e \in H_i^j$,

$$cost^*(T, e) = \frac{path(T, e)}{w(e)} \leq 2y^{j-i+2}. \tag{9}$$

To see this, note that since e is covered during iteration j, by the previous lemma $path(T, e) \leq 2y^{j+1}$, while on the other hand, $e \in E_i$ implies $w(e) > y^{i-1}$.

By Lemma 5.4,

$$|H_i^j| \leq |E_i^j| \leq |E_i|/x^{j-i}. \tag{10}$$

It follows from inequalities (9) and (10) that for every $1 \leq i \leq j$,

$$cost^*(T, H_i^j) \leq 2x^2\mu^{j-i+2}|E_i|. \tag{11}$$

Summing these costs over $i \leq j \leq i+\rho-1$, we get that

$$cost^*(T, E_i) \leq \sum_{j=i}^{i+\rho-1} cost^*(T, H_i^j) \leq 4x^2\mu^{\rho+1}|E_i|$$

for every $i \geq 1$. Summing the costs over all weight classes we get

$$cost^*(T, E) = \sum_{i=1} cost^*(T, E_i) \leq 4x^2\mu^{\rho+1}|E|,$$

and the lemma follows. □

It follows that the average spanning factor of the constructed tree T is bounded above by

$$S(G, w, T) \leq 4x^2\mu^{\rho+1} \leq 4x^2\left(\frac{27\ln^2 n}{\ln x}\right)^{1+\lceil \frac{3\ln n}{\ln x}\rceil}.$$

Selecting $x = \exp(\sqrt{\log n \log\log n})$ optimizes this bound on $S(G, w, T)$ as $S(G, w, T) \leq \exp(O(\sqrt{\log n \log\log n}))$. This completes the proof of Theorem 5.3. □

COROLLARY 5.7. *There exists a constant c such that for n sufficiently large, every n vertex connected multigraph G, w satisfies $Val^*(G, w) \leq \exp(c\sqrt{\log n \log\log n})$.*

Proof. By Theorem 4.1, $Val^*(G, w) = \max_{G',w'} S_{opt}(G', w')$, where G', w' ranges over all the weighted multigraphs obtained from G, w by replication. The claim thus follows from Theorem 5.3. □

We observe that the tree construction algorithm as described above can be used within a procedure (based on the ellipsoid algorithm) for implementing the first step in the online k-server algorithm of Theorem 3.1, yielding a polynomial-time algorithm for the problem; i.e., the optimal strategy for the tree player can, in fact, be efficiently approximated.

6. Grids and hypercubes. In this section, we discuss the value of the game on unweighted grids and hypercubes.

6.1. Upper bounds for d-dimensional grids. We consider the d-dimensional grid $G_{n,d}$ that is the Cartesian product of d n-vertex paths and has $N = n^d$ vertices. Formally, the vertex set of $G_{n,d}$ consists of all vectors of length d whose coordinates are in $\{1, \ldots, n\}$. The grid contains an edge uv if the vectors u and v differ in exactly one coordinate where their values differ by exactly 1.

A hypercube is edge-transitive, so $Val^*(G_{2,d}) = S_{opt}(G_{2,d})$. For $G_{n,2}$, the value of the game can be approximated by the optimal tree. To see this, note that the n by n discrete torus is edge-transitive. Any tree in the grid G is also a tree in the torus G'. If we choose a tree T in G with diameter less than αn, then $S(G', T) \leq S(G, T)(1 - \frac{1}{n}) + \alpha$. On the other hand, $Val^*(G') \geq Val^*(G)$, because the edge player on the torus has the option of playing only edges in the grid. The upper bound we obtain is independent of d.

THEOREM 6.1. *For the d-dimensional grid $G_{n,d}$ with $N = n^d$ vertices, $S_{opt}(G_{n,d}) < 2 \log N$.*

Proof. Suppose $n = 2^k$. For the upper bound, we construct a tree T_k in $G_{n,d}$ of diameter

$$d_k \leq (2d - 1)(2^k - 1) = (2d - 1)(n - 1)$$

such that $S(G_{n,d}, T_k) < 2d \log n = 2 \log N$. Partition the vertices of $G_{n,d}$ into 2^d subsets inducing copies of $G_{n/2,d}$. There is a unique d cube Q consisting of one vertex from each of these subsets. Form T_k by using a copy of T_{k-1} in each of the 2^d subsets and a spanning tree of diameter $2d - 1$ in Q (a breadth-first search tree from any vertex will do). Figure 2 illustrates the construction for $G_{8,2}$. We have $d_k \leq (2d - 1) + 2d_{k-1}$, with $d_0 = 0$. Hence $d_k \leq (2d - 1)(2^k - 1)$. Note that $G_{n,d}$ has $dn^{d-1}(n - 1) = dN(1 - \frac{1}{n})$ edges; let $N' = (\frac{n}{2})^d$. Also, the number of nontree edges that join the copies of $G_{n/2,d}$ is $dn^{d-1} - (2^d - 1)$. We thus have

$$S(G_{n,d}, T_k) \leq \frac{2^d \cdot dN'(1-\frac{2}{n}) S(G_{n/2,d}, T_{k-1}) + (dn^{d-1} - 2^d + 1)(d_k + 1)}{dN(1-\frac{1}{n})}$$

$$J < S(G_{n/2,d}, T_{k-1}) + \frac{(d_k+1)}{n-1} < S(G_{n/2,d}, T_{k-1}) + 2d.$$

Hence $S(G_{n,d}, T_k) < 2dk$. □

For the interesting special case of the hypercube $Q_d = G_{2,d}$, which is edge-transitive, we have a slightly better construction. This construction improves on the general case only by roughly a factor of four, but it is still of interest, since it is conjectured to be optimal.

THEOREM 6.2. $Val^*(Q_d) = S_{opt}(Q_d) \leq (d + 1)/2$.

Proof. Equality holds for $d = 1, 2$, with no choice of tree. For $d > 2$, construct a tree T_d by taking the optimal $d - 1$-dimensional tree T^* in copy A of Q_{d-1} and adding all edges between this and the other copy of Q_{d-1}. The edges in copy A have the same $cost^*$ as before,

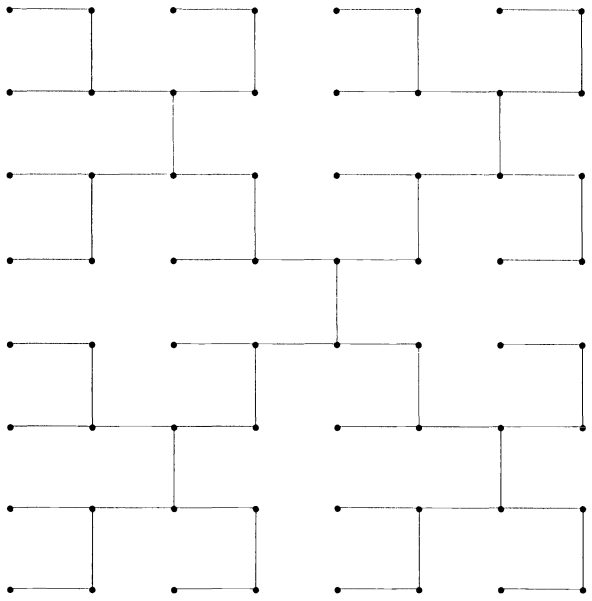


FIG. 2. *A spanning tree in* $G_{8,2}$.

and those between the copies have $cost^* = 1$. Each edge e in copy B corresponding to an edge e' in copy A has $cost^*(T_d, e) = cost^*(T^*, e') + 2$. We have

$$\begin{aligned} S_{opt}(Q_d) &\leq S(Q_d, T_d) \\ &= \frac{1}{(d2^{d-1})}\left((d-1)2^{d-2}(2 \cdot S_{opt}(Q_{d-1}) + 2) + 2^{d-1} \cdot 1\right). \end{aligned}$$

Hence $S_{opt}(Q_d) < \frac{d-1}{d} S_{opt}(Q_{d-1}) + 1$, implying the claim inductively. □

We believe that the general construction described above is essentially optimal for large n, with $S_{opt}(G_{n,d}) = 2\lg N + o(\lg N)$. In the following sections we prove only that $S_{opt}(G_{n,d}) \geq c \log N$ for some constant c.

6.2. Lower bounds for two-dimensional grids. For clarity, it is useful to first present the lower bound for the case of $d = 2$, showing $S_{opt}(G_{n,2}) \geq c\lg n + o(\lg n)$. For this we need several preliminary results.

LEMMA 6.3. *If A is a set of at least α^2 vertices in $G_{n,2}$, where $|A| \leq \frac{n^2}{2}$, then there are at least α rows that A intersects but does not fill or at least α columns that A intersects but does not fill.*

Proof. Suppose A intersects r rows and s columns, with $r \geq s$. Then $rs \geq \alpha^2$ implies $r \geq \alpha$, and the result holds for A unless A fills at least one row. If A does fill a row then $n = s \leq r$. Thus A intersects every row and every column. Now our goal is to prove that there are at least α rows that A does not fill or at least α columns that A does not fill. For this not to happen means that A fills more than $n - \alpha$ rows and more than $n - \alpha$ columns. This requires that A has more than $n^2 - \alpha^2$ vertices, which is impossible if $\alpha^2 \leq |A| \leq \frac{n^2}{2}$. □

LEMMA 6.4. *If A is a set of at least α^2 vertices in $G_{n,2}$, where $|A| \leq \frac{n^2}{2}$, and B is a set of at most four vertices in A, then there are at least $\frac{\alpha}{2}$ vertices in A that have neighbors outside A and have distance at least $\frac{\alpha}{16}$ from each vertex of B.*

Proof. The preceding lemma guarantees a set C of at least α vertices in A that are in distinct rows (or distinct columns) and have neighbors outside A. Since these are in distinct

rows, a vertex of B can be within distance $\frac{\alpha}{16}$ of at most $\frac{\alpha}{8}$ vertices in C. When we delete from C the vertices that are too close to B, at least $\frac{\alpha}{2}$ vertices remain. □

LEMMA 6.5. *If T is a spanning tree of $G_{n,2}$ and $1 \leq \alpha \leq \frac{n}{16}$, then there are at least $\frac{n^2}{64\alpha}$ edges e such that $path(T, e) > \frac{\alpha}{16}$.*

Proof. Since the grid has maximum degree 4, every subtree on $m > 1$ vertices in it (not necessarily spanning) has an edge whose deletion separates it into two smaller subtrees, each having at least $\frac{m-1}{4} \geq m/8$ of its vertices. Begin with T and successively delete edges from the biggest remaining component, always splitting that component as evenly as possible. Do this until $\lceil \frac{n^2}{8\alpha^2} \rceil - 1$ edges have been deleted and the number of pieces is $\lceil \frac{n^2}{8\alpha^2} \rceil$. On the average, the pieces have about $8\alpha^2$ vertices, with the smallest piece having at least α^2 vertices. To see this, note that at every stage, the largest piece has at most 8 times as many vertices as the smallest piece, because whenever there is a new smallest size it is at least $\frac{1}{8}$ of the old largest size. Minimizing x_1 subject to $x_1 \leq \cdots \leq x_m \leq 8x_1$ and $\sum x_i = M$ yields $x_1 = \frac{M}{8m-3}$, achieved when all the other variables are 8 times the smallest. Note also that the largest piece must have fewer than half the vertices as soon as more than 16 pieces are requested.

Since each deleted edge of T is incident to two pieces, the average number of deleted edges incident with a piece is less than 2. Hence at least half the pieces are incident with at most four deleted edges. (Since otherwise the number of deleted edges would be too large.) If A is the set of vertices of such a piece, let B be the vertices in A incident to these deleted edges. By the preceding lemma, A has at least $\frac{\alpha}{2}$ vertices having neighbors outside A whose distance from each vertex of B is at least $\frac{\alpha}{16}$. The edges leaving A from these vertices do not belong to T, since the only edges of T leaving A are incident to vertices of B. For any such edge e, we have $path(T, e) > \frac{\alpha}{16}$, since the path in T between the ends of e must contain a path in the piece of T induced by A from e to one of the vertices in B. Harvesting at least $\frac{\alpha}{2}$ endpoints of such edges from each of at least $\frac{n^2}{16\alpha^2}$ pieces of T (i.e., those pieces incident with at most 4 deleted edges), yields at least $\frac{n^2}{64\alpha}$ such edges, since we might have counted each edge twice. (Of course, $path(T, e) > \frac{\alpha}{8}$ for any edge we have counted twice, but we are not trying to optimize the constants here.) □

We are now ready to prove our lower bound.

THEOREM 6.6. $S_{opt}(G_{n,2}) \geq \frac{1}{2048} \ln n - O(1)$.

Proof. Given an arbitrary spanning tree T of $G_{n,2}$, the preceding lemma guarantees at least $\frac{n^2}{32\alpha}$ edges with $cost^*$ at least $\frac{\alpha}{16}$ for any $\alpha \leq \frac{n}{16}$. We now choose an edge e at random, defining a random variable X by $X = cost^*(T, e)$. We seek a lower bound on $E[X]$, the average $cost^*$ of an edge, since $S_{opt}(G_{n,2}) = E[X]$. We use the fact that, for a discrete random variable X attaining nonnegative integer values, we have $E[X] = \sum_{k \geq 1} Prob(X \geq k)$. For $k \leq \frac{n}{256}$, let $\alpha = 16k$. Then

$$Prob(X \geq k) \geq \frac{n^2}{1024k \cdot 2n(n-1)} > \frac{1}{2048}.$$

We thus obtain

$$E[X] > \frac{1}{2048} \sum_{k=1}^{\lfloor n/256 \rfloor} \frac{1}{k} \geq \frac{1}{2048} \ln n - O(1). \qquad \square$$

6.3. Lower bounds for d-dimensional grids. More generally, we prove that Theorem 6.1 is optimal, up to a constant factor, for all $n, d \geq 2$.

THEOREM 6.7. *There exists an absolute constant $c > 0$ such that for every $n, d \geq 2$, $S_{opt}(G_{n,d}) \geq cd \log n = c \log N$.*

The proof of Theorem 6.7 requires several preparations. We prove it for, say, $c = 10^{-10}$. (We note that we make no attempt to optimize the constants here and in the rest of the proof.) Clearly, with this value of c, if both n and d do not exceed 10^5 there is nothing to prove and hence we may assume that at least one of them does exceed 10^5, i.e.,

$$\max\{n, d\} \geq 10^5. \tag{12}$$

Suppose $n, d \geq 2$ and let $G = G_{n,d} = (V, E)$. Denote $M = \{1, 2, \ldots, n\}$. Recall that the vertices of G are naturally represented by all vectors of length d whose coordinates are in M. If uv is an edge of G, where $u, v \in V$, we say that the *direction* of the edge uv is i if i is the unique coordinate in which u and v differ. In the course of the proof we need to consider vectors whose coordinates except one are all determined. It is convenient to denote a nondetermined coordinate by the symbol $*$. Thus, for a subset $A \subset V$ and for a vector $v = (v_1, \ldots, v_d)$ with $v_i = *$ and $v_j \in M$ for all $j \neq i$, we say that A *intersects* the vector v if there is a $u \in A$ such that u coincides with v on all the coordinates but the ith (i.e., the projection of u on $\{1, \ldots, d\} \setminus \{i\}$ is equal to that of v). We say that A *properly intersects* v if A intersects v and also its complement $V \setminus A$ intersects v. Note that in this case there is at least one edge uw of G in direction i which joins a vertex u of A with a vertex w of $V \setminus A$, with the property that both u and w coincide with v on all coordinates but the ith.

We need the following lemma, proved in [CFGS]. (The proof in [CFGS] is given only for the case $n = 2$, but the same proof works for all integers n. See also [Al].)

LEMMA 6.8 [CFGS]. *Let $D = \{1, \ldots d\}$ be a finite set and let $B_1, \ldots, B_m$ be subsets of D such that each element of D belongs to at least k of the sets B_i. Let $\mathcal{F}$ be a family of vectors of length d whose coordinates, indexed by the elements of D, lie in $M = \{1, \ldots, n\}$. For each i, $1 \leq i \leq m$, let $\mathcal{F}_i$ denote the set of all projections of the vectors in $\mathcal{F}$ on B_i. Then*

$$|\mathcal{F}|^k \leq \prod_{i=1}^{m} |\mathcal{F}_i|. \qquad \square$$

Returning to the graph $G = (V, E)$ we now prove the following lemma.

LEMMA 6.9. *Suppose $A \subset V$, $|A| = x^d$, where $x \geq 1$ is not necessarily an integer. Let A_i denote the set of all vectors v such that A intersects v, where $v_i = *$ and $v_j \in M$ for all $j \neq i$. Let A_i' denote the set of all vectors v as above such that A properly intersects v. Then*

$$\sum_{i=1}^{d} |A_i| \geq dx^{d-1}$$

and

$$\sum_{i=1}^{d} |A_i'| \geq dx^{d-1}(1 - x/n).$$

Proof. Apply Lemma 6.8 to the grid $G_{n,d} = (V, E)$, where D is the set of d coordinates in the vectors of V. Specifically, set $\mathcal{F} = A$, $m = d$, $k = d - 1$ and $B_i = D \setminus \{i\}$. Clearly in this case $|\mathcal{F}_i| = |A_i|$ and hence, by the lemma

$$x^{d(d-1)} = |A|^{d-1} \leq \prod_{i=1}^{d} |A_i| \leq \left(\frac{1}{d} \sum_{i=1}^{d} |A_i| \right)^d.$$

This implies the first part of Lemma 6.9.

To prove the second part, observe that since $|A| = x^d$, A cannot intersect without properly intersecting more than x^d/n vectors v whose ith coordinate is $*$. Thus $|A_i'| \geq |A_i| - x^d/n$ for all i, and summation over $1 \leq i \leq d$ completes the proof of the lemma. □

Let v be a vertex of G, let y be a positive real, and suppose $i \in D = \{1, \ldots, d\}$. Let S denote the set of all vertices of G whose distance from v is less than $y(d-1)$ and let S_i denote the set of projections of the members of S on $D \setminus \{i\}$. Denote by $L(y, d)$ the maximum possible cardinality of S_i where the maximum is taken over all possible choices of v and i.

LEMMA 6.10. *For all* $y > 0$

$$L(y, d) \leq 2^{Min\{d-1, y(d-1)\}} \binom{d-1+(d-1)y}{d-1}. \tag{13}$$

Therefore, for all $y \geq 1$,

$$L(y, d) \leq (2e(y+1))^{d-1}$$

and for all $y \leq 1$

$$L(y, d) \leq (2e^2)^{(d-1)y},$$

where $e = 2.71828\ldots$ *is the basis of the natural logarithm.*

Proof. Suppose $v = (v_1, \ldots, v_d)$ and let F be a set of vertices of G whose distance from v is at most $y(d-1)$. Suppose, further, that no two members of F have the same projection on $D \setminus \{i\}$. Clearly, it suffices to show that the cardinality of F is at most the right-hand side of (13). If $u = (u_1, \ldots, u_d)$ is a member of F then, for each $j \neq i$, $u_j = v_j + \epsilon_j$, where $\sum_{j \neq i} |\epsilon_j| < y(d-1)$. The number of ways to choose the signs of all the ϵ_j-s which are not 0 is at most $2^{\min\{d-1, y(d-1)\}}$, since there are at most $\min\{d-1, y(d-1)\}$ such ϵ_j. The number of ways to choose the absolute values of the ϵ_js is less than the number of ways to write $y(d-1)$ as an ordered sum of d nonnegative integers (the first $d-1$ of which will serve as our numbers ϵ_j), and this is precisely the binomial coefficient $\binom{d-1+(d-1)y}{d-1}$. This completes the proof of (13). The other two estimates follow from this one by using the fact that $\binom{a}{b} \leq (ea/b)^b$ for all integers a and b and the fact that $(1 + \frac{1}{y})^y \leq e$. □

We can now return to the proof of Theorem 6.7. Let T be a spanning tree of $G = G_{n,d}$. Let x be a real number, $1.5 \leq x \leq \frac{3}{4}n$. By deleting edges of T we can break it into connected components each containing at most x^d and at least $\frac{x^d}{4d}$ vertices. (This is possible since the maximum degree of a vertex of G (and hence of a vertex of T) is at most $2d$, and thus if we repeatedly split T into connected components by always splitting the biggest remaining component as evenly as possible we will never have two components the ratio between the sizes of which exceeds $4d$.) It follows that the number t of connected components, and hence the number of deleted edges of T, does not exceed $\frac{4dn^d}{x^d}$. Let U denote the set of all end vertices of these deleted edges. Then $|U| \leq \frac{8dn^d}{x^d}$.

Let x_j denote the positive dth root of the number of vertices in the jth connected component, $(1 \leq j \leq t)$. Since $x_j \leq x \leq 0.75n$ for each j, Lemma 6.9 implies that the total number of edges of G emanating from the jth connected component to vertices in other components is at least $\frac{1}{4}dx_j^{d-1}$. Observe that if y is a real positive number and if an endpoint of such an edge is at distance at least $y(d-1)$ from all the vertices in U that lie in the same connected component as that endpoint then the length of the elementary cycle corresponding to this edge is greater than $y(d-1)$. This is because the elementary cycle has to contain a vertex in U. We next obtain a lower bound for the number of such edges. Let us fix a direction i and consider only edges in direction i. If we let A denote the set of vertices of the jth component C_j and

use the notation of Lemma 6.9, we conclude that there are at least $|A_i'|$ vertices in C_j whose projections on $D \setminus \{i\}$ are pairwise different, such that from each such vertex there is an edge in direction i emanating to another connected component. By the definition of $L(y, d)$, each vertex of U can be at distance less than $y(d-1)$ from no more than $L(y, d)$ such vertices. Summing over all t connected components and over all d directions we conclude that the number of edges that join distinct components whose elementary cycles have length that exceeds $y(d-1)$ is at least

$$\frac{1}{2}\sum_{j=1}^{t}\frac{1}{4}dx_j^{d-1} - |U|dL(y, d).$$

Since $|U| \leq 8dn^d/x^d$ and since the minimum possible value of $\sum_{j=1}^{t} x_j^{d-1}$ subject to the constraint that $x \geq x_j \geq 0$ and $\sum_{j=1}^{t} x_j^d = n^d$ is $\frac{n^d}{x^d}x^{d-1} = n^d/x$ we obtain the following lemma.

LEMMA 6.11. *For each spanning tree T in $G = G_{n,d}$ and for each real x, $1.5 \leq x \leq 0.75n$ and positive real y, the number of edges of G whose elementary cycles have length that exceeds $y(d-1)$ is at least*

$$\frac{dn^d}{8x} - L(y, d)8d^2\frac{n^d}{x^d}. \qquad \square$$

We can now prove Theorem 6.7. For technical reasons it is convenient to consider three possible cases, depending on the values of the parameters n and d.

Case I. $n \leq 80$.

In this case $d \geq 10^5$, by (12). Put $x = 1.5$, $y = 1/20$. By Lemma 6.10, $L(y, d) \leq (2e^2)^{0.05(d-1)} < 1.2^{d-1}$. Hence (since $x = 1.5$)

$$L(y, d)8d^2\frac{n^d}{x^d} < \left(\frac{12}{15}\right)^{d-1} 8d\frac{dn^d}{x} < \frac{dn^d}{16x}.$$

Therefore, by Lemma 6.11, there are at least $\frac{dn^d}{16x} = \frac{1}{24}dn^d$ edges whose cycles have length that exceeds $\frac{1}{20}(d-1)$, and since the total number of edges is $dn^{d-1}(n-1) < dn^d$ and the tree T was arbitrary we conclude that in this case $S_{opt}(G_{n,d}) \geq \frac{1}{480}(d-1) > cd\log n$, as needed.

Case II. $d \leq 20$.

In this case, which requires a little more work, $n \geq 10^5$, by (12). For each real x satisfying

$$1600 \leq x \leq \frac{3}{4}n, \tag{14}$$

define $y = x/1600$. By Lemma 6.10, $L(y, d) \leq (2e(y+1))^{d-1} < \frac{x^{d-1}}{160^{d-1}}$ and hence

$$L(y, d)8d^2\frac{n^d}{x^d} \leq \frac{8d}{160^{d-1}}\frac{dn^d}{x} \leq \frac{dn^d}{10x},$$

where here we applied the fact that $d \geq 2$.

It follows from Lemma 6.11 that at least a fraction $1/(40x)$ of the edges have cycles of length more than $x(d-1)/1600$, for each x satisfying (14). Therefore, if X is the random variable defined on the edges f of our graph G by letting $X(f)$ be the length of the elementary cycle of f divided by $d-1$, it follows that for each x satisfying (14), the probability $Prob(X \geq x/1600)$ is at least $1/(40x)$. Therefore, by defining $z = x/1600$ we conclude that for each

integer z satisfying $1 \leq z \leq \frac{3}{6400}n$, $Prob(X \geq z) \geq 1/(64000z)$. Thus the expectation of X is at least

$$\sum_{z=1}^{\lfloor 3n/6400 \rfloor} Prob(X \geq z) \geq \sum_{z=1}^{\lfloor 3n/6400 \rfloor} \frac{1}{64000z} \geq 10^{-6} \log n.$$

The average cost of an edge with respect to the tree T is $d-1$ times the expectation of X and hence we conclude that in this case $S_{opt}(G_{n,d}) \geq 10^{-6}(d-1)\log n > cd \log n$, as required.

Case III. $n > 80$ and $d > 20$.

For each real x, $30 \leq x \leq 0.75n$ define y by $y+1 = \frac{x}{4e}$. By Lemma 6.10, $L(y,d) \leq \frac{x^{d-1}}{2^{d-1}}$ and hence

$$L(y,d)8d^2\frac{n^d}{x^d} \leq \frac{8d}{2^{d-1}}\frac{dn^d}{x} < \frac{dn^d}{16x}.$$

Therefore, by Lemma 6.11, for every x, $30 \leq x \leq 0.75n$, with probability at least $1/(16x)$, the random variable X defined in the previous case exceeds $(x-4e)/(4e)$. As before (by letting $z = \lceil (x-4e)/4e \rceil$ take integer values) it follows that the expectation of this random variable is at least

$$\sum_{z \geq \lceil 30/(4e)-1 \rceil}^{\lceil 3n/(16e)-1 \rceil} \frac{1}{64e(z+1)} > 10^{-6} \log n,$$

and this implies that in this case too $S_{opt}(G_{n,d}) \geq cd \log n$ and completes the proof of Theorem 6.7. □

REFERENCES

[ADDJ] I. Althöfer, G. Das, D. Dobkin, and D. Joseph, *Generating sparse spanners for weighted graphs*, in Proc. 2nd Scandinavian Workshop on Algorithm Theory, Springer-Verlag, New York, July 1990, pp. 26–37.

[Al] N. Alon, *Probabilistic methods in extremal finite set theory*, Proc. of the Conf. on Extremal Problems for Finite Sets, Bolyai Society Mathematical Studies, Hungary, 1991.

[Aw] B. Awerbuch, *Complexity of network synchronization*, J. Assoc. Comput. Mach., 32 (1985), pp. 804–823.

[AwPe] B. Awerbuch and D. Peleg, *Sparse Partitions*, Proc. 31st Annual Symp. on Foundations of Computer Science, IEEE Press, New York, pp. 503–513, 1990.

[BBKTW] S. Ben-David, A. Borodin, R.M. Karp, G. Tardos, and A. Wigderson, *On the power of randomization in online algorithms*, in Proc. 22nd Annual ACM Symp. on Theory of Computing, pp. 379–386, 1990.

[Bo] B. Bollobás, *Extremal Graph Theory*, Academic Press, New York, 1978.

[BLS] A. Borodin, N. Linial, and M. Saks, *An optimal on-line algorithm for metrical task systems*, J. Assoc. Comput. Mach. to appear.

[Cai] L. Cai, *Tree Spanners: Spanning Trees that Approximate Distances*, Ph.D. Thesis, Tech. Report 260/92, Univ. of Toronto, Toronto, Canada, May 1992.

[CFGS] F. R. K. Chung, P. Frankl, R. L. Graham, and J. B. Shearer, *Some Intersection Theorems for Ordered Sets and Graphs*, J. Combinatorial Theory Ser. A, 43 (1986), pp. 23–37.

[ChLa] M. Chrobak and L. Larmore, *An Optimal On-Line Algorithm for k Servers on Trees*, SIAM J. Comput., 20 (1991), pp. 144–148.

[FiRaRa] A. Fiat, Y. Rabani, and Y. Ravid, *Competitive k-server Algorithms*, Proc. 31st Annual IEEE Symp. on Foundations of Computer Science, pp. 454–463, 1990.

[FRRS] A. Fiat, Y. Rabani, Y. Ravid, and B. Schieber, *A deterministic $O(k^3)$–competitive k-server algorithm for the circle*, Algorithmica, 11 (1994), pp. 572–578.

[FKLMSY] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young, *Competitive Paging Algorithms*, J. Algorithms, 12 (1991), 685-699.

[Hu] T. C. Hu, *Optimum Communication Spanning Trees*, SIAM J. Comput., 3 (1974), pp. 188–195.

[JLR] D. S. Johnson, J. K. Lenstra, and A. H. G. Rinnooy Kan, *The Complexity of the Network Design Problem*, Networks, 8 (1978), pp. 279–285.

[MaMcSl] M. S. Manasse, L. A. McGeoch, and D. D. Sleator, *Competitive Algorithms for On-Line Problems*, J. Algorithms, 11 (1990), pp. 208–230.

[PS] D. Peleg and A. A. Schäffer, *Graph Spanners*, J. Graph Theory, 13 (1989), pp. 99–116.

[PU] D. Peleg and J. D. Ullman, *An optimal synchronizer for the hypercube*, SIAM J. Comput., 18 (1989), pp. 740–747.

[SV] J. M. Stern and S. A. Vavasis, *Nested dissection for Sparse Nullspace Bases*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 766–775.

SIAM J. COMPUT.
Vol. 24, No. 1, pp. 101–108, February 1995

006

FINDING k CUTS WITHIN TWICE THE OPTIMAL*

HUZUR SARAN† AND VIJAY V. VAZIRANI†

Abstract. Two simple approximation algorithms for the minimum k-cut problem are presented. Each algorithm finds a k cut having weight within a factor of $(2 - 2/k)$ of the optimal. One algorithm is particularly efficient—it requires a total of only $n - 1$ maximum flow computations for finding a set of near-optimal k cuts, one for each value of k between 2 and n.

Key words. graph partitioning, minimum cuts, approximation algorithms

AMS subject classifications. 68Q20, 68Q25

1. Introduction.

The minimum k-cut problem is as follows: Given an undirected graph $G = (V, E)$ with nonnegative edge weights and a positive integer k, find a set $S \subseteq E$ of minimum weight whose removal leaves k connected components. This problem is of considerable practical significance, especially in the area of VLSI design. Solving this problem exactly is NP hard [GH], but no efficient approximation algorithms were known for it.

In this paper we give two simple algorithms for finding k cuts. We prove a performance guarantee of $(2 - 2/k)$ for each algorithm; however, neither algorithm dominates the other in all instances. One of our algorithms is particularly fast; it requires only $n - 1$ max flow computations, using the classic result of [GoHu]. In fact, within the same running time, this algorithm finds near-optimal k cuts for each k, $2 \leq k \leq n$. We also give a family of graphs that show that the bound of $(2 - 2/k)$ is tight for each algorithm. The problem of achieving a factor of $(2 - \epsilon)$, for some fixed $\epsilon > 0$, independent of k, seems to be difficult, and may possibly be intractable.

The minimum k-cut problem and its variants have been extensively studied in the past. For fixed k, polynomial time algorithms for solving the problem exactly have been discovered by [DJPSY] for planar graphs and by [GH] for arbitrary graphs. These algorithms have running times of $O(n^{ck})$, for some constant c, and $O(n^{k^2})$, respectively. More efficient algorithms for the case of planar graphs and $k = 3$ are given in [He] and [HS]. Polynomial time algorithms have also been developed for several variants [Ch], [Cu], and [Gu].

In [DJPSY], the complexity of multiway cuts is studied: Given an edge-weighted undirected graph with k specified vertices, find a minimum weight k cut that separates these vertices. They show that this problem is NP hard even when k is fixed, for $k \geq 3$. They also give an approximation algorithm that finds a solution within a factor of $(2 - 2/k)$ of the optimal, using k max-flow computations. Using their algorithm as a subroutine one can get a $(2 - 2/k)$ approximation algorithm for our problem; however this will require $\binom{n}{k}$ calls and is therefore not polynomial time in case k is not fixed.

We shall first present the more efficient algorithm, which we call EFFICIENT. The other algorithm is called SPLIT. We will actually establish the $(2 - 2/k)$ performance guarantee for a slight variant of EFFICIENT. The weight of the k cut found by this variant dominates the k cuts found by both EFFICIENT and SPLIT. Finally, we report on some preliminary results obtained by applying these techniques to the balanced graph partitioning problem.

2. Algorithm EFFICIENT.

Let $G = (V, E)$ be a connected undirected graph and let $wt : E \rightarrow \mathbf{Z}^+$ be an assignment of weights to the edges of G. We will extend the function wt to subsets of E in the obvious manner. A partition $(V', V - V')$ of V specifies a *cut*; the cut consists of all edges, S, that have one endpoint in each partition. We will denote a cut by the

*Received by the editors August 7, 1992; accepted for publication (in revised form) May 28, 1993.
†Department of Computer Science and Engineering, Indian Institute of Technology, New Delhi-110016, India.

set of its edges, S, and will define its *weight* to be $wt(S)$. For any set $S \subseteq E$, we will denote the number of connected components of the graph $G' = (V, E - S)$ by comps(S).

We will first find a k cut for a specified k. Our algorithm is based on the following greedy strategy: It keeps picking cuts until their union is a k cut and in each iteration it picks the lightest cut; in the original graph, that creates additional components.

Algorithm EFFICIENT:

1. For each edge e, pick a minimum weight cut, say, s_e, that separates the endpoints of e.
2. Sort these cuts by increasing weight, obtaining the list $s_1, \ldots, s_m$.
3. Greedily pick cuts from this list until their union is a k cut; cut s_i is picked only if it is not contained in $s_1 \cup \cdots \cup s_{i-1}$.

(Note: in case the algorithm ends with a cut B such that comps(B) $> k$, we can easily remove edges from B to get a cut B' such that comps(B') $= k$.)

Notice that since edge e is in s_e, $s_1 \cup \cdots \cup s_m = E$, and so this must be an n cut. Therefore, the algorithm will certainly succeed in finding a k cut.

Let $b_1, \ldots, b_l$ be the successive cuts picked by the algorithm. In the next lemma we show that with each cut picked we increase the number of components created, and hence the total number of cuts picked is at most $k - 1$.

LEMMA 2.1. *For each i, $1 \leq i < l$,* comps($b_1 \cup \cdots \cup b_{i+1}$) $>$ comps($b_1 \cup \cdots \cup b_i$).

Proof. Because of the manner in which EFFICIENT picks cuts, $b_{i+1} \not\subseteq (b_1 \cup \cdots \cup b_i)$. Let (u, v) be an edge in $b_{i+1} - (b_1 \cup \cdots \cup b_i)$. Clearly, u and v are in the same component in the graph obtained by removing the edges of $b_1 \cup \cdots \cup b_i$ from G, and they are in different components in the graph obtained by removing $b_1 \cup \cdots \cup b_{i+1}$ from G. Hence, the latter graph has more components. □

COROLLARY 2.2. *The number of cuts picked, l, is at most $k - 1$.*

Notice that at each step we are indeed picking the lightest cut that creates additional components: among all edges that lie within connected components, we choose the edge whose endpoints can be disconnected with the lightest of the cuts from the original graph.

The complexity of our algorithm is dominated by the time taken to find cuts $s_1, \ldots, s_m$. This can clearly be done with m max flow computations. A more efficient implementation is obtained by using Gomory–Hu cuts. Gomory and Hu [GoHu] show that there is a set of $n - 1$ cuts in G such that for each pair of vertices, $u, v \in V$, the set contains a minimum weight cut separating u and v; moreover, they show how to find such a set using only $n - 1$ max flow computations. The cuts $s_1, \ldots, s_m$ can clearly be obtained from such a set of $n - 1$ cuts.

Incorporating all this, we get a particularly simple description of our algorithm. First, notice that step 3 is equivalent to

3′. Find the minimum i such that comps($s_1 \cup \ldots \cup s_i$) $\geq k$. Output the k cut $s_1 \cup \ldots \cup s_i$.

Next, observe that when implemented with Gomory–Hu cuts, Algorithm EFFICIENT essentially boils down to the following:

1. Find a set of Gomory-Hu cuts in G.
2. Sort these cuts by increasing weight, obtaining the list $g_1, \ldots, g_{n-1}$.
3. Find the minimum i such that comps($g_1 \cup \ldots \cup g_i$) $\geq k$.

The algorithm extends in a straightforward manner to obtaining near-optimal k cuts for each k, $2 \leq k \leq n$, with the same set of Gomory–Hu cuts.

3. Performance guarantee for EFFICIENT. In this section we shall prove the next theorem.

THEOREM 3.1. *Algorithm EFFICIENT finds a k cut having weight within a factor of* $(2 - 2/k)$ *of the optimal.*

Let $b_1, \ldots, b_l$ be the successive cuts found by Algorithm EFFICIENT; $B = b_1 \cup \cdots \cup b_l$. Central to our proof is a special property of these cuts with respect to an enumeration of all cuts in G. Let $c_1, c_2, \ldots$ be such an enumeration, sorted by increasing weight, such that $b_1, \ldots, b_l$ appear in this order in the enumeration. Such an enumeration will be said to be *consistent* with the cuts $b_1, \ldots, b_l$. For any cut c in G, index(c) is its index in this enumeration.

The union property. Let $s_1, \ldots, s_p$ be a set of cuts in G, sorted by increasing weight. Pick any enumeration of all cuts in G, $c_1, c_2, \ldots$ that is consistent with $s_1, \ldots, s_p$. We will say that $s_1, \ldots, s_p$ satisfy the union property if, intuitively, w.r.t. any such enumeration, the union of the cuts in any initial segment of $c_1, c_2, \ldots$ is equal to the union of all cuts s_j contained in this initial segment. More formally, pick any consistent enumeration of all cuts in G, let i be any index, $1 \leq i \leq \text{index}(s_p)$, and let s_q be the last cut in the sorted order having index at most i. Then, $c_1 \cup \cdots \cup c_i = s_1 \cup \cdots \cup s_q$.

LEMMA 3.2. *The cuts* $b_1, \ldots, b_l$ *satisfy the union property.*

Proof. Suppose the cuts do not satisfy the union property. Then there is an enumeration of all cuts in G, $c_1, c_2, \ldots$, that is consistent with $b_1, \ldots, b_l$, and yet there is an index i, $i \leq \text{index}(b_l)$, such that

$$b_1 \cup \cdots \cup b_q \neq c_1 \cup \cdots \cup c_i,$$

where b_q is the last cut on our list having index at most i. The smallest such index will be referred to as the *point of discrepancy*. Let us fix an enumeration for which the point of discrepancy is maximum. Clearly, the point of discrepancy will have index less than index(b_l).

Let b_{q+1} be the next cut picked by our algorithm, and let index$(b_{q+1}) = j$. Because of the manner in which we fixed the enumeration, $wt(c_j) > wt(c_i)$ (otherwise we could interchange c_i and c_j, and obtain an enumeration in which discrepancy occurs at a larger index).

Clearly, $b_q \neq c_i$ and $c_1 \cup \cdots \cup c_{i-1} = b_1 \cup \cdots \cup b_q$. Let $e \in c_i - (c_1 \cup \cdots \cup c_{i-1})$. Then, $e \notin b_1 \cup \cdots \cup b_q$. Clearly, $wt(s_e) \leq wt(c_i)$, where s_e is a minimum weight cut that disconnects the endpoints of e. Now, our algorithm must pick edge e with a cut of weight at most $wt(s_e)$, and hence at most $wt(c_i)$. Since $wt(b_{q+1}) > wt(c_i)$, this means that $e \in b_1 \cup \cdots \cup b_q$, leading to a contradiction. □

Remark. Since $b_1, \ldots, b_l$ are drawn from $s_1, \ldots, s_m$, the latter cuts also satisfy the union property. For similar reasons, $g_1, \ldots, g_{n-1}$ satisfy the union property as well.

Let A be a minimum weight k cut in G. The second key idea in our proof is to view A as the union of k cuts as follows: Let $V_1, \ldots, V_k$ be the connected components of $G' = (V, E - A)$. Let a_i be the cut that separates V_i from $V - V_i$ for $1 \leq i \leq k$. Then $A = \bigcup_{i=1}^{k} a_i$. Notice that $\sum_{i=1}^{k} wt(a_i) = 2wt(A)$ (because each edge of A is counted twice in the sum). Assume without loss of generality that $wt(a_1) \leq wt(a_2) \leq \cdots \leq wt(a_k)$.

The $(2 - 2/k)$ bound is established by showing that the sum of weights of our cuts, $b_1, \ldots, b_l$ is at most the sum of weights of $a_1, \ldots, a_{k-1}$, i.e.,

$$\begin{aligned} wt(B) &\leq \sum_{i=1}^{l} wt(b_i) \leq \sum_{i=1}^{k-1} wt(a_i) \\ &\leq 2(1 - 1/k)wt(A), \end{aligned}$$

since a_k is the heaviest cut of A.

Actually, it will be simpler to prove a stronger statement. We will consider a slight variant of EFFICIENT that picks cuts with multiplicity; cut b_i will be picked t times if its inclusion created t additional components, i.e., if $(\text{comps}(b_1 \cup \ldots \cup b_i) - \text{comps}(b_1 \cup \ldots \cup b_{i-1})) = t$.

So, now we have exactly $k-1$ cuts; let us call them $b_1, \ldots, b_{k-1}$, to avoid introducing excessive notation. As before, $b_1, \ldots, b_{k-1}$ are sorted by increasing weight, and moreover we shall assume that cuts with multiplicity occur consecutively. We will show that

$$\sum_{i=1}^{k-1} wt(b_i) \leq \sum_{i=1}^{k-1} wt(a_i) \ldots (i).$$

For the rest of the proof, we will study how the cuts a_i's and b_j's are distributed in $c_1, c_2, \ldots$, an enumeration of all cuts in G w.r.t. which $b_1 \ldots b_{k-1}$ satisfy the union property. Denote by α_i the number of all cuts a_j that have index $\leq i$, i.e., $\alpha_i = |\{a_j | \text{ index}(a_j) \leq i\}|$. We will show that for each index i, $1 \leq i \leq \text{index}(a_{k-1})$, the number of cuts b_j having index $\leq i$ is at least α_i (of course, cuts b_j are counted with multiplicity). If so, there will be a 1–1 map from $\{b_1, \ldots b_{k-1}\}$ onto $\{a_1, \ldots, a_{k-1}\}$ such that if b_i is mapped onto a_j, then index $(b_i) \leq \text{index}(a_j)$. This will prove (i).

Two nice properties of the cuts a_i's and b_j's will help prove the assertion of the previous paragraph. Denote by A_i the union of all cuts a_j that have index $\leq i$, i.e., $A_i = \bigcup_{a_j:\text{index}(a_j)\leq i} a_j$. Similarly, let $B_i = \bigcup_{b_j:\text{index}(b_j)\leq i} b_j$.

The next lemma shows that for each index i, the cuts b_j are making at least as much progress as the cuts a_j's, where progress is measured by the number of components created.

LEMMA 3.3. *For each index i, $\text{comps}(A_i) \leq \text{comps}(B_i)$.*

Proof. The lemma is clearly true for $i > \text{index}(b_{k-1})$. For $i \leq \text{index}(b_{k-1})$, $B_i = c_1 \cup \cdots \cup c_i$, since the cuts $b_1, \ldots, b_{k-1}$ satisfy the union property. Therefore, $A_i \subseteq B_i$, and the lemma follows. □

It is easy to construct an example showing that each cut a_j may not necessarily create additional components. Yet, at each index i, the number of components created by the cuts a_j is at least $\alpha_i + 1$. This is established in the next lemma.

LEMMA 3.4. *For each i, $1 \leq i \leq \text{index}(a_{k-1})$, $\text{comps}(A_i) \geq \alpha_i + 1$.*

Proof. Corresponding to each cut a_j having index $\leq i$, the partition V_j will be a single connected component all by itself in the graph $G_i = (V, E - A_i)$. Let us charge a_j to this component of G_i. Since index $(a_k) > i$, the component of G_i containing V_k will not get charged. The lemma follows. □

At this point we have all the ingredients to finish the proof. Consider an index i, $1 \leq i \leq \text{index}(a_{k-1})$. By Lemma 3.4, $\text{comps}(A_i) \geq \alpha_i + 1$. This together with Lemma 3.3 gives $\text{comps}(B_i) \geq \alpha_i + 1$. For each additional component created by us, we have included a cut b_j (by including cuts with appropriate multiplicity), and thus it follows that the number of cuts b_j having index $\leq i$ is at least α_i. The theorem follows.

Remark. The proof given above shows that any set of cuts satisfying the union property will give a near-optimal k cut. This explains why the Gomory–Hu cuts work.

Clearly, the proof given above holds *simultaneously* for each value of k between 2 and n. Hence we get a set of near-optimal k cuts, $2 \leq k \leq n$. Notice that at the extremes, i.e., for $k = 2$ and $k = n$, we get optimal cuts.

THEOREM 3.5. *Algorithm EFFICIENT finds a set of k cuts, one for each value of k, $2 \leq k \leq n$; each cut is within a factor of $(2 - 2/k)$ of the optimal k cut. The algorithm requires a total of $n - 1$ max flow computations.*

Using the current best known max flow algorithm [GT], [KRT], our algorithm has a running time of $O(mn^2 + n^{3+\epsilon})$, for any fixed $\epsilon > 0$.

4. Algorithm SPLIT. Perhaps the first heuristic that comes to mind for finding a k cut is the following.

Algorithm SPLIT: Start with the given graph. In each iteration, pick the lightest cut in the current graph that splits a component, and remove the edges of this cut. Stop when the current graph has k connected components.

Notice that SPLIT, like EFFICIENT, is also a greedy algorithm. Whereas EFFICIENT picks a lightest cut in the original graph that creates additional components, SPLIT picks a lightest cut in the current graph.

SPLIT needs to find a minimum weight cut in each new component formed—this can be done using $n - 1$ max flow computations in a graph on n vertices. Hence SPLIT requires $O(kn)$ max flow computations to find a k cut.

We shall establish a $(2 - 2/k)$ performance guarantee for SPLIT as well. However, first let us point out that neither algorithm dominates the other. Consider the following graph on eight vertices, $\{a, b, c, d, e, f, g, h\}$. The edges and their weights are as follows.

$wt(a, b) = 3$,
$wt(a, c) = 3$,
$wt(b, d) = 7$,
$wt(c, e) = 7$,
$wt(d, e) = 10$,
$wt(d, f) = 4$,
$wt(f, g) = 5$,
$wt(g, h) = 5$,
$wt(e, h) = 4$.

Now, for $k = 3$, the cuts found by SPLIT and EFFICIENT have weights 13 and 14, respectively, but for $k = 4$, the weights are 20 and 19, respectively.

THEOREM 4.1. *The k cut found by Algorithm SPLIT has weight within a factor of $(2-2/k)$ of the optimal.*

Proof. In Theorem 3.1 we showed that a slight variant of EFFICIENT, that picks cuts with appropriate multiplicity, has a performance bound of $(2 - 2/k)$. We will now prove that the weight of the k cut found by SPLIT is dominated by the weight of the k cut found by this variant.

Let $b_1, \ldots, b_{k-1}$ be the cuts picked by the variant. Notice that since SPLIT picks a minimum weight cut in a component, it splits it into exactly two components. Therefore, SPLIT picks exactly $k - 1$ cuts, say, $d_1, \ldots, d_{k-1}$.

By induction on i, we will show that $wt(d_i) \leq wt(b_i)$ for $1 \leq i \leq k - 1$. The assertion is clearly true for $i = 1$. To show the induction step, first notice that $\text{comps}(d_1 \cup \cdots \cup d_i) = i + 1$ and $\text{comps}(b_1 \cup \cdots \cup b_{i+1}) \geq i + 2$. Therefore, there is a cut b_j, $1 \leq j \leq i + 1$, that is not contained in $(d_1 \cup \cdots \cup d_i)$. By the proof of Lemma 2.1, this cut will create additional components, and is available to SPLIT. Hence, SPLIT will pick a cut of weight at most $wt(b_j)$. Since $wt(b_j) \leq wt(b_{i+1})$, the assertion follows. □

5. Lower bound. We will show that the bounds established in Theorem 3.1 and Theorem 4.1 are tight in the following sense.

THEOREM 5.1. *For any ϵ, $0 < \epsilon \leq 1$, there is an infinite family of minimum k-cut instances (G, wt, k) such that the weight of the k cut found by each algorithm, EFFICIENT and SPLIT, lies in the range*

$$[(1 - \epsilon)(2 - 2/k)W,\ (2 - 2/k)W],$$

where W is the weight of an optimal k cut for the instance. Moreover, k is unbounded in this family.

Proof. The kth instance, in which we want to find a k cut, consists of a graph on $2k-1$ vertices, $V = \{u_1, \ldots, u_{k-1}, v_1, \ldots, v_k\}$. The only edges are (with weights specified)

$$\begin{aligned} wt(u_i, u_{i+1}) &= \beta \quad \text{for } 1 \le i \le k-2, \\ wt(u_{k-1}, v_1) &= \beta, \\ wt(v_i, v_{i+1}) &= \alpha \quad \text{for } 1 \le i \le k-1, \\ wt(v_1, v_k) &= \alpha, \end{aligned}$$

where α is a positive integer, and $\beta = 2\alpha(1-\epsilon)$.

The minimum k cut, A, picks all edges of weight α, and so $wt(A) = k\alpha$. Each algorithm, EFFICIENT and SPLIT, picks the k cut, B, consisting of all edges of weight β. So $wt(B) = (k-1)\beta = 2(k-1)\alpha(1-\epsilon)$. Hence

$$wt(B)/wt(A) = (2 - 2/k)(1-\epsilon).$$

This proves the theorem. □

6. Balanced graph partitioning. Given an edge-weighted graph, $G = (V, E)$, $wt : E \to \mathbf{Z}^+$, with an even number of vertices, the balanced graph partitioning problem asks for a partition of V into two sets, V_1 and V_2, each containing half the vertices, such that the weight of the cut separating V_1 and V_2 is minimized. This problem is NP hard [GJ]. It models realistic situations such as circuit partitioning and is frequently used in practice. (See [KL] for the well-known "swap" heuristic for this problem.) We give the first approximation algorithm for this problem; it achieves a performance ratio of $n/2$. The algorithm extends in a straightforward manner to the problem of partitioning the graph into k equal size pieces, for any fixed k. The performance ratio achieved for this problem is $(\frac{k-1}{k})n$. In the past, [LR] and [LMPSTT] have used multicommodity flow for obtaining a $\frac{1}{3}, \frac{2}{3}$ graph partition algorithm that is within an $O(\log n)$ factor of the best balanced partition (in a $\frac{1}{3}, \frac{2}{3}$ graph partition each side of the cut has between $\frac{1}{3}$ and $\frac{2}{3}$ fraction of the vertices).

Our algorithm uses the fact that there is a pseudopolynomial time algorithm for determining whether n given numbers $a_1, \ldots, a_n$ can be partitioned into two sets each of which adds up to $W/2$, where $\sum_{i=1}^{n} a_i = W$. This algorithm is based on a straightforward dynamic programming approach, and has a running time of $O(nW)$ [GJ]; if the answer is "yes," it finds a valid partition as well.

Balanced graph partitioning algorithm:

1. Find a set of Gomory-Hu cuts in G.
2. Sort these cuts by increasing weight, obtaining $g_1, \ldots, g_{n-1}$.
3. Find the minimum i such that the connected components of $G' = (V, E - (g_1 \cup \cdots \cup g_i))$ can be partitioned into two sets containing $\frac{n}{2}$ vertices each.

The complexity of our algorithm is dominated by step 1, since the total time required by step 3 is $O(n^3)$. We will use the following property of the partitioning problem to establish the bound of $n/2$.

LEMMA 6.1. *Let n be an even integer, and let $a_1, \ldots a_{(n/2)+1}$ be positive integers such that*

$$\sum_{i=1}^{\frac{n}{2}+1} a_i = n.$$

Then the answer to the partitioning problem is "yes."

Proof. Without loss of generality assume that the a_i's are sorted in decreasing order. Notice that in order to maintain a sum of n, if $a_1 = 2$ then $a_2 = \cdots = a_{(n/2)-1} = 2$ and $a_{n/2} = a_{(n/2)+1} = 1$. In this case, the a_i's can clearly be partitioned. In general, let k be the largest index such that $a_k > 1$, i.e., the last $\frac{n}{2} + 1 - k$ numbers are 1's. It is easy to see that the sizes of a_i's exceeding 2 determine the number of 1's in the following manner:

$$\sum_{i=1}^{k}(a_i - 2) = \left(\frac{n}{2} + 1 - k\right) - 2. \tag{1}$$

Now, associate $a_i - 2$ distinct 1's with each a_i that exceeds 2. By (1) this is feasible, and moreover exactly two 1's will be left over unassociated. For each a_i exceeding 2, we will include a_i in one partition and its associated 1's in the other partition. This effectively gives the former partition a weight of 2 more than the latter. Hence, once again we are essentially left with partitioning numbers of the form $2, 2, \ldots, 2, 1, 1$. □

As before, let $c_1, c_2, \ldots$ be an enumeration of all cuts in G, ordered by increasing weight, and let $B = g_1 \cup \cdots \cup g_i$ be the set of edges picked by our algorithm. Let c_k be the first cut in the enumeration that yields an optimal balanced partitioning of G.

LEMMA 6.2. $wt(B) \leq \frac{n}{2} wt(c_k)$.

Proof. As in Algorithm EFFICIENT, among the cuts $g_1, \ldots, g_i$, pick g_j iff it is not contained in $(g_1 \cup \cdots \cup g_{j-1})$. Let $b_1, \ldots, b_l$ be the cuts picked in this manner. Clearly, $B = b_1 \cup \cdots \cup b_l$. The proof is based on Lemma 3.2, i.e., the fact that the cuts $b_1, \ldots, b_l$ satisfy the union property. This and the fact that the components of $G' = (V, E - C)$ can certainly be partitioned into equal sized sets, for any set C containing c_k, imply that index$(b_l) \leq k$. Now by Lemma 2.1, $l \leq n - 1$, thereby giving a factor of $(n - 1)$. To achieve a better factor, notice that Lemma 6.1 implies that $l \leq n/2$. Therefore $wt(B) \leq \sum_{i=1}^{l} wt(b_i) \leq \frac{n}{2} wt(c_k)$. □

LEMMA 6.3. *The bound of $\frac{n}{2}$ is tight for our algorithm.*

Proof. As in Theorem 5.1, for any ϵ, $0 < \epsilon < 1$, we will give an infinite family of instances on which the cut found by our algorithm lies in the range $[(1 - \epsilon)\frac{n}{2} W, \frac{n}{2} W]$, where W is the weight of the optimal cut. The kth instance consists of a graph on $2k$ vertices $\{u, v, u_1 \ldots u_{k-1}, v_1, \ldots, v_{k-1}\}$, and edges (with weights) $wt(u_i, u) = \alpha, 1 \leq i \leq k - 1$, $wt(v_i, v) = \alpha, 1 \leq i \leq (k - 1)$, $wt(u, v) = \beta$, where $\alpha = (1 - \epsilon)\beta$. It is easy to see that our algorithm finds a cut of weight $k\alpha$, and the optimal cut has weight β. □

THEOREM 6.4. *The algorithm given above finds a balanced partitioning of an edge-weighted graph, using $n - 1$ max-flow computations. The weight of the cut found is within a factor of $n/2$ of the optimal. Moreover, the bound of $n/2$ is tight for our algorithm.*

7. Discussion and open problems. It will be interesting to see how Algorithms SPLIT and EFFICIENT compare in practice, even though neither algorithm dominates the other in worst-case performance. Our guess is that SPLIT will typically give lighter k cuts.

Notice that the graphs given in Theorem 5.1 help establish the $(2 - 2/k)$ lower bound for the k cut found for each k, $2 \leq k \leq (n + 1)/2$, where n is the number of vertices in the graph (n is odd), but not for higher values of k. Certainly, the bound of $(2 - 2/k)$ is not tight for $k = n$, since we get the optimal cut. Is there a better analysis of our algorithms for $k > n/2$?

Is there a better approximation algorithm for the minimum k-cut problem? We believe that the problem of achieving a factor of $(2 - \varepsilon)$, for some $\varepsilon > 0$, independent of k, is intractable. The minimum k-cut problem is MAX-SNP complete [Pa], and hence by the recent result of [ALMSS], achieving a factor of $1 + \varepsilon$, is NP complete for some $\varepsilon > 0$.

Our investigation of the balanced graph partitioning problems appears to be quite preliminary, and it should be possible to improve the bound. Interesting special cases are (a) the graph is planar, and (b) all edge weights are 1. Since the graphs used in Lemma 6.3 are planar,

the lower bound for our algorithm holds for case (a). It is easy to see that if ties are resolved arbitrarily in our algorithm, Lemma 6.3 holds for case (b) as well; however, this seems to be a small hurdle. It will also be useful to consider the version of the balanced graph partitioning problem in which vertices have weights; in this case the pseudopolynomial algorithm for partition will not be useful.

Finally, some of these methods may be useful for obtaining approximation algorithms for other NP-hard graph partitioning problems (see [GH] and [GJ]).

Acknowledgments. We wish to thank Fan Chung, Samir Khuller, Laszlo Lovasz, Milena Mihail, Christos Papadimitriou, Umesh Vazirani, and Mihalis Yannakakis for valuable discussions and comments.

REFERENCES

[ALMSS] S. ARORA, C. LUND, R. MOTWANI, M. SUDAN, AND M. SZEGEDY, *Proof verification and intractability of approximation problems*, Proc. 33rd Annual Symp. on the Foundations of Computer Science, 1992, pp. 14–23.

[Ch] V. CHVÁTAL, *Tough graphs and Hamiltonian circuits*, Discrete Math., 5 (1973), pp. 215–228.

[Cu] W. H. CUNNINGHAM, *Optimal attack and reinforcement of a network*, J. Assoc. Comput. Mach., 32 (1985), pp. 549–561.

[DJPSY] E. DALHAUS, D. S. JOHNSON, C. H. PAPADIMITRIOU, P. SEYMOUR, AND M. YANNAKAKIS, *The complexity of the multiway cuts*, Proc. 24th Annual ACM Symposium on the Theory of Computing, 1992, pp. 241–251.

[GoHu] R. GOMORY AND T. C. HU, *Multi-terminal network flows*, J. SIAM, 9 (1961), pp. 551–570.

[GH] O. GOLDSCHMIDT AND D. S. HOCHBAUM, *Polynomial algorithm for the k-cut problem*, Proc. 29th Annual Symp. on the Foundations of Computer Science, 1988, pp. 444–451.

[GJ] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-completeness*, Freeman, San Francisco, CA, 1979.

[GT] A. V. GOLDBERG AND R. E. TARJAN, *A new approach to the maximum flow problem*, Proc. of the 18th Annual ACM Symp. on Theory of Computing, 1987, pp. 136–146.

[Gu] D. GUSFIELD, *Connectivity and edge-disjoint spanning trees*, Inform. Process. Lett., 16 (1983), pp. 87–89.

[He] X. HE, *On the planar 3-cut problem*, J. Algorithms, 12 (1991), pp. 23–37.

[HS] D. HOCHBAUM AND D. SHMOYS, *An $O(V^2)$ algorithm for the planar 3-cut problem*, SIAM J. Discrete Meth., 6:4:707–712, 1985.

[KL] B. W. KERNIGHAN AND S. LIN, *An efficient heuristic for partitioning graphs*, Bell Systems Technical Journal, 40 (1970), pp. 291–308.

[KRT] V. KING, S. RAO, AND R. TARJAN, *A faster deterministic maximum flow algorithm*, Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms, 1992, pp. 157–164.

[LMPSTT] T. LEIGHTON, F. MAKEDON, S. PLOTKIN, C. STEIN, E. TARDOS, AND S. TRAGOUDAS, *Fast approximation algorithms for multicommodity flow problems*, Proc. 23rd Annual ACM Symp. on the Theory of Computing, 1991, pp. 101–111.

[LR] T. LEIGHTON AND S. RAO, *An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms*, Proc. 29th Annual Symp. on the Foundations of Computer Science, 1988, pp. 422–431.

[Pa] C. PAPADIMITRIOU, private communication, 1991.

SIAM J. COMPUT.
Vol. 24, No. 1, pp. 109–122, February 1995

007

THE SERIAL TRANSITIVE CLOSURE PROBLEM FOR TREES*

MARIA LUISA BONET† AND SAMUEL R. BUSS†

Abstract. The serial transitive closure problem is the problem, given a directed graph G and a list of edges, called closure edges, which are in the transitive closure of the graph, to generate all the closure edges from edges in G. A nearly linear upper bound is given on the number of steps in optimal solutions to the serial transitive closure problem for the case of graphs that are trees. "Nearly linear" means $O(n \cdot \alpha(n))$, where α is the inverse Ackermann function. This upper bound is optimal to within a constant factor.

Key words. transitive closure, graph algorithm, inverse Ackermann function, weak superconcentrators

AMS subject classifications. 68R10, 05C12, 05C85

1. Introduction.

A directed graph is transitive if, whenever there is an edge from a node X to a node Y and an edge from Y to Z, there is an edge from X to Z. The transitive closure of G is a smallest transitive, directed graph containing G. We write $X \rightarrow Y$ to indicate the presence of an edge from X to Y. It is easy to see that any edge in the transitive closure of a graph G can be obtained from the edges of G by a series of zero or more *closure steps*, which are inferences of the form

$$\frac{A \rightarrow B \qquad B \rightarrow C}{A \rightarrow C}.$$

In other words, if $A \rightarrow B$ and $B \rightarrow C$ are edges in the transitive closure of G, then $A \rightarrow C$ is too. This is because G plus the edges that can be derived by closure steps from edges in G both must be in any transitive graph containing G and also form a transitive graph on the nodes of G.

The serial transitive closure problem is the problem of deriving a given subset of edges, which we call "closure edges," in the transitive closure of a directed graph. A solution to the serial transitive closure problem is a sequence of closure steps that generates all of the given closure edges and the size of a solution is the number of closure steps in the solution. In this paper, we give upper and lower bounds on the size of optimal solutions to the serial transitive closure problems for directed graphs that are trees. It should be stressed that the set of closure edges can be any subset of the edges in the transitive closure of the graph (but not in the graph). The degenerate case of deriving a single closure edge $A \rightarrow B$ is quite simple, since the minimum number of closure steps required will be one less than the length of a shortest path from A to B. The general question of determining the optimal size of a solution is made difficult by the fact that, when a *set* of closure edges is being derived, it may be possible for individual closure steps to aid in the generation of multiple closure edges. In other words, it is not necessary to generate each closure edge independently. It is also important that the set of closure edges will, in general, not be *all* the edges in the transitive closure of the graph; the problem of deriving all the edges in the transitive closure of the graph is uninteresting because, in this case, exactly one closure step is needed per closure edge.

The authors' interest in the serial transitive closure problem arose in the study of the lengths of propositional proofs with and without the deduction rule. The serial transitive closure problem is directly related to the question of how efficiently Frege proof systems can simulate nested deduction Frege proof systems. More information on these proof systems and

*Received by the editors January 27, 1992; accepted for publication (in revised form) September 14, 1993.

†Department of Mathematics, University of California San Diego, La Jolla, California 92903. This research was supported in part by National Science Foundation grant DMS-8902480.

the application of the serial transitive closure problem to proof lengths can be found in [2], [4], [3]. The present paper is an expansion of portions of [2], [4].

The serial transitive closure problem is potentially applicable to problems in networks. As an example, suppose a communication network is given where the nodes are, say, computers, and an edge from X to Y indicates that X is capable of sending messages to Y. Correspondingly, a set of closure edges is a set of edges of desired connections. If the closure edges are in the transitive closure of the network, then it is possible to establish the desired connections by having nodes relay messages (much like real-world nets such as Usenet). A closure step would correspond to determining that, since X can (indirectly) transmit to Y and Y can (indirectly) transmit to Z, X can indirectly transmit to Z. The size of a solution to the serial transitive closure problem would correspond to the number of indirect communication links that must be established to set up the desired connections. (It should be mentioned that this example completely glosses over important issues such as the bandwidth of the connections.)

The serial transitive closure problem is formally defined as follows.

Serial Transitive Closure Problem.

An *instance* consists of

- A directed graph G with m edges and
- A list of n *closure* edges $X_i \to Y_i$ $(i = 1, \ldots, n)$ that are in the transitive closure of G but not in G.

A *solution* is a sequence of edges $U_i \to V_i$ $(i = 1, \ldots, s)$ containing all n closure edges such that each $U_i \to V_i$ is inferred by a single closure step from earlier edges in the solution and/or edges in G. We call s the number of steps of the solution.

Note that the number of steps in a solution counts only closure steps and does not count edges that are already in G. A directed graph is a *tree* if it is a tree in the usual sense, with root at the top and with all edges directed downward or with all edges directed upward.

The outline of this paper is as follows. In §2, we state and prove the main results, which give near-linear upper bounds on size of solutions to the serial transitive closure problem for trees. Our near-linear upper bounds are of the form $O(n \cdot \alpha(n))$, where $\alpha(n)$ is the extremely slow-growing inverse Ackermann function. In §3 we show that the upper bound is optimal for trees, using a theorem used by Tarjan [13] for an algorithm for the Union-Find problem. It is not known whether our upper bound is optimal for the case where the directed graph G is linear, i.e., G is a tree with each nonleaf node having only one child. However, we argue at the end of §3 that our construction cannot be easily improved for the linear case, since our approach gives an explicit construction for weak superconcentrators and there is a lower bound on the size of constant depth superconcentrators [10] that prevents the possibility that a simple modification can lead to an improvement of our construction.

The methods of our paper do not apply to graphs that are not trees, and we do not know any nontrivial upper or lower bounds on the size of solutions of serial transitive closure problems for general graphs.

Our proof methods for our upper bounds (Theorem 2 through Corollary 2.8) are related to the constructions of weak superconcentrators by [6], [7], and are also similar to prior methods for creating algorithms for range queries on linear lists [15] and on free trees [8]. In fact, as one of the referees pointed out, it is possible to derive Corollary 2.8, as a corollary to the proofs (but not the theorems) contained in [15], [8]. Similar constructions have also been used for adding edges to trees to reduce their diameter [1]. However, it is useful to give direct proofs in this paper since the serial transitive closure problem is of independent interest (e.g., as applied in [4], [3]).

2. Upper bounds for trees. The "near-linear" upper bounds given below are stated in terms of extremely slow growing functions such as log^* and the inverse Ackermann function. The function $\log x$ is the real-valued base two logarithm function. The log^* function is defined so that $\log^* n$ is equal to the least number of iterations of the logarithm base 2, which, applied to n, yields a value < 2. In other words, $\log^* n$ is equal to the least value of k such that $n < 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$ where there are k 2's in the stack. To get even slower growing functions, we define the $\log^{(*i)}$ functions for each $i \geq 0$. The $log^{(*0)}$ function is just the base 2 logarithm function rounded down to an integer and the $log^{(*1)}$ is just the log^* function. For $i > 1$, $log^{(*i)}(n)$ is defined to be equal to the least number of iterations of the $log^{(*i-1)}$ function, which, applied to n, yields a value < 2. The Ackermann function can be defined by the equations:

$$\begin{aligned} A(0, m) &= 2m, \\ A(n+1, 0) &= 1, \\ A(n+1, m+1) &= A(n, A(n+1, m)). \end{aligned}$$

It is well known that the Ackermann function is recursive but dominates eventually every primitive recursive function (see [9] for a proof). We next develop the basic properties of the Ackermann function and the $\log^*$ functions; see La Poutré [12] for a similar development (his function $\alpha(i, m)$ is equal to our $\log^{(*i-1)}(m)$).

It is is easy to see, by induction on n, that $A(n, 1) = 2$ for all n, because

$$A(n+1, 1) = A(n, A(n+1, 0)) = A(n, 1).$$

Also, by induction on m, we have $A(1, m) = 2^m$, since

$$A(1, m+1) = A(0, A(1, m)) = 2 \cdot A(1, m).$$

Likewise, $A(2, m) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$ where there are m 2's in the stack, since

$$A(2, m+1) = A(1, A(2, m)) = 2^{A(2,m)}.$$

PROPOSITION 2.1. *For $n > 1$, $A(n, m)$ is the equal to the least i such that $\log^{(*n-1)}(i) = m$. Hence, $\log^{(*n-1)} A(n, m) = m$.*

Proof. The proof is by induction on n. By the above definitions and comments, the lemma holds for $n = 1, 2$. Fix n and assume, as the induction hypothesis for n, that for all i, m, if $A(n, m) \leq i < A(n, m+1)$, then $\log^{(*n-1)}(i) = m$. To prove the corresponding fact for $n+1$, we use induction on m. It is obvious for $m = 0$ since $A(n, 0) = 0$ and $A(n, 1) = 2$. For $m > 0$, we have that

$$\begin{aligned} \log^{(*n-1)} A(n+1, m) &= \log^{(*n-1)} A(n, A(n+1, m-1)) \\ &= A(n+1, m-1) \end{aligned}$$

and, in addition, that

$$\begin{aligned} \log^{(*n-1)}\Big(A(n+1, m+1) - 1\Big) &= \log^{(*n-1)}\Big(A(n, A(n+1, m)) - 1\Big) \\ &= A(n+1, m) - 1. \end{aligned}$$

The last equality is justified by the induction hypothesis that $A(n, A(n+1, m))$ is the least value i such that $\log^{(*n-1)}(i) = A(n+1, m)$. Thus we have shown that, if $A(n+1, m) \leq$

$k < A(n+1, m+1)$, then $A(n+1, m-1) \leq \log^{(*n-1)}(k) < A(n+1, m)$. If follows that, if $A(n+1, m) \leq k < A(n+1, m+1)$, then $\log^{(*n-1)}$ must be applied exactly m times to k to yield a value $< 2 = A(n+1, 1)$; i.e., that $\log^{(*n)}(k) = m$. □

PROPOSITION 2.2. *For $i \geq 1$, $\log^{(*i-1)}(i) \leq 3$.*

Proof. First observe that for all $i \geq 0$ and $x \geq 1$, we have $\log^{(*i)}(x) < x$. And for $i \geq 0$, $\log^{(*i)}(3) < 2$. We now prove the proposition by induction on i. The base case, $i = 1$, is obvious. For the induction step, we have that $\log^{(*i-1)}(i+1) \leq i$ by the first observation,and $\log^{(*i-1)}(i) \leq 3$ by the induction hypothesis. Thus three applications of $\log^{(*i-1)}$ suffice to take $i+1$ to a value less than 2. Hence $\log^{(*i)}(i+1) \leq 3$. □

DEFINITION. *The* inverse Ackermann *function α is defined so that $\alpha(n)$ is equal to the least value of i such that $A(i, i) > n$. Equivalently, $\alpha(n)$ is equal to the least i such that $\log^{(*i-1)} n < i$.*

MAIN THEOREM 2.3. *Let $i \geq 0$. If the directed graph G is a tree then the serial transitive closure problem has a solution with $O(n + m \log^{(*i)} m)$ steps.*

MAIN THEOREM 2.4. *If the directed graph G is a tree then the serial transitive closure problem has a solution with $O((n+m) \cdot \alpha(m))$ steps.*

Another definition of the inverse Ackermann function has been given by Tarjan [13] who defines

$$\alpha(m, n) = \text{least } i \geq 1 \text{s.t.} A(i, 4\lceil m/n \rceil) > \log n.$$

We will also prove below that for G a tree, the serial transitive closure has a solution with $O((n+m)\alpha(n+m, m))$ steps.

Theorem 2.7 below is a restatement of Theorem 2 with explicit constants. The rest of our upper bounds will be corollaries of Theorem 2.7.

For the proofs of our main theorems we may assume without loss of generality that G is a rooted tree with edges pointing away from the root. We always picture trees with the root at the top, except in the special case of one-trees, which have fanout 1, the root is to the left and edges point to the right. The concepts of *child, father, ancestor*, and *descendent* are defined as usual. The *size* of a tree is defined to be the number of edges in the tree (not the number of nodes). If a tree has e edges, then it has exactly $e+1$ nodes. We define a *subtree* of a tree T to be any connected subset of the edges of T and their endpoints so that, for all nodes X and Y in T, if X and Y have the same father, then X is in the subtree iff Y is in the subtree. A subtree is *unscarred* if it consists of all the edges below a given node in the tree. A subtree S may also be obtained by first removing some set of unscarred subtrees and then letting S consist of all the remaining edges below some given remaining node: S is said to have a *scar* at any of its leaf nodes that are roots of earlier removed (nontrivial) subtrees. Two subtrees are said to be *disjoint* if they have no edges in common; disjoint subtrees may share a single node since the root of one may be a scar of the other. If X is a node in T, then T_X denotes the subtree of T rooted at X. The *immediate subtrees* of a tree T are the maximal proper subtrees of T, i.e., the trees T_X for X a child of the root of T. The following lemma is well known: see, e.g., Brent [5].

LEMMA 2.5. *Let $N \geq 0$ and T be a tree with $\geq N$ edges. Then there is an unscarred subtree of T that has size $\geq N$ edges such that each of its immediate subtrees has $< N$ edges.*

Lemma 2.5 is easily proved by taking a minimal subtree with $\geq N$ edges. The next theorem restates and strengthens the case $i = 0$ of Theorem 2.3 with fairly tight bounds on the constants. The *log* function is base 2.

THEOREM 2.6. *If the directed graph G is a tree then the serial transitive closure problem has a solution with $n + m \cdot \lfloor \log m \rfloor$ closure steps.*

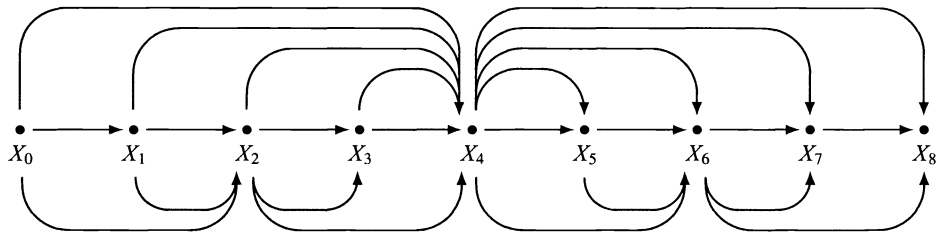


FIG. 1.

Proof. We will first derive $\leq m\lfloor \log m \rfloor$ edges, called *auxiliary edges*; each auxiliary edge will be obtained with a single closure step. The choice of auxiliary edges is independent of the closure edges; however, from the edges in G and the auxiliary edges each closure edge can be obtained with (at most) one additional closure step.

For illustration purposes, we first prove the theorem for G a one-tree and then do the general case. Although the general case includes the linear case, the proof of the linear case presents the main ideas more clearly.

Linear case. Assume G is a one-tree; that is, each node except the leaf has a single child. In this case we may assume the nodes of G are named $X_0, \ldots, X_m$ and that the edges of G are just $X_i \to X_{i+1}$ for $0 \leq i < m$. The auxiliary edges will be derived in rounds, the first round will in essence split G into two subtrees of $m/2$ edges, the second round splits G into four subtrees of $m/4$ edges, etc., for a total of $\lceil \log m \rceil - 1$ rounds. The process is illustrated for the case $m = 8$ in Fig. 1; the upper edges of Fig. 1 are derived in the first round and the lower edges in the second round.

Round 1. $X_{\lfloor m/2 \rfloor}$ is the midpoint of the one-tree G. The auxiliary edges added in round 1 are the edges of the form $X_j \to X_{\lfloor m/2 \rfloor}$ for $0 \leq j < \lfloor m/2 \rfloor$ and the edges of the form $X_{\lfloor m/2 \rfloor} \to X_k$ for $m/2 < k \leq m$. There are exactly m such edges and they can be derived with m closure steps if we derive them in the right order; namely, letting j range from $\lfloor m/2 \rfloor - 1$ down to 0 and letting k range from $\lfloor m/2 \rfloor + 1$ up to m. (Actually only $m - 2$ closure steps are needed since two of the auxiliary edges are already in G.)

Round 2. Round 1 split G into two halves; the midpoints of these two halves are $X_{\lfloor m/4 \rfloor}$ and $X_{\lfloor 3m/4 \rfloor}$. In round two, auxiliary edges to and from these midpoints are derived. Namely, (1) the edges $X_j \to X_{\lfloor m/4 \rfloor}$ with $j < \lfloor m/4 \rfloor$; (2) the edges $X_{\lfloor m/4 \rfloor} \to X_j$ with $\lfloor m/4 \rfloor < j \leq \lfloor m/2 \rfloor$; (3) the edges $X_j \to X_{\lfloor 3m/4 \rfloor}$ with $\lfloor m/2 \rfloor \leq j < \lfloor 3m/4 \rfloor$; and (4) the edges $X_{\lfloor 3m/4 \rfloor} \to X_j$ with $\lfloor 3m/4 \rfloor < j \leq m$. There are m such edges, and by deriving them in the right order, each can be obtained with a single closure step. (Again, taking into account duplicate edges, fewer than m closure steps are needed for round 2.)

Round ℓ. For round number ℓ we add auxiliary edges incident on the nodes $X_{\lfloor k \cdot m/2^\ell \rfloor}$ for odd values of k. Specifically, for each odd value $k < 2^\ell$ the following auxiliary edges are derived: (1) the edges $X_j \to X_{\lfloor k \cdot m/2^\ell \rfloor}$ for $\lfloor (k-1)m/2^\ell \rfloor \leq j < \lfloor k \cdot m/2^\ell \rfloor$ and (2) the edges $X_{\lfloor k \cdot m/2^\ell \rfloor} \to X_j$ for $\lfloor k \cdot m/2^\ell \rfloor < j \leq \lfloor (k+1)m/2^\ell \rfloor$. Again, there are exactly m such edges (some of them duplicates of edges from G and edges from earlier rounds); this is seen by using the obvious one-to-one correspondence with the edges of G. And by deriving them in the right order, each auxiliary edge is obtained with a single closure step.

Since there are exactly $\lceil \log m \rceil - 1$ rounds and fewer than m closure steps are needed in each round, it is clear that there are $\leq m\lfloor \log m \rfloor$ auxiliary edges and that the number of closure steps so far is bounded by $m\lfloor \log m \rfloor$.

Now we claim that each of the n closure edges can be obtained with at most a single closure step from the $m\lfloor \log m \rfloor$ auxiliary edges (of course some of the closure edges may also be auxiliary edges). To prove this, suppose $X_i \to X_j$ is a closure edge; of course, $i+1 < j$. Find the least value of ℓ such that for some odd k, $i \leq \lfloor k \cdot m/2^\ell \rfloor \leq j$. If either of the inequalities are actually equalities, then $X_i \to X_j$ is an auxiliary edge added in round ℓ and no additional closure step is needed. If both inequalities are strict, then $X_i \to X_{\lfloor k \cdot m/2^\ell \rfloor}$ and $X_{\lfloor k \cdot m/2^\ell \rfloor} \to X_j$ are both auxiliary edges and from these the closure edge $X_i \to X_j$ can be derived with one closure step.

It follows that all the closure and auxiliary edges are derived with fewer than $n + m\lfloor \log m \rfloor$ closure steps and Theorem 2.6 is proved for G a one-tree.

General case. The proof of Theorem 2.6 for G a tree uses a construction similar to the proof of the linear case. For the general case, we use Lemma 2.5 to split G into multiple subtrees of size less than half the size of G (one of these is scarred); then we similarily split these subtrees into subtrees of size less than one quarter the size of G, etc. As in the linear case, we derive auxiliary edges of G as we split G into subtrees; this process will be done in $\leq \log m$ rounds.

Round 1. By Lemma 2.5 there is a node X in G such that G_X has $\geq m/2$ edges, but the immediate subtrees of G_X have size $< m/2$ edges. Let $G_1, \ldots, G_k$ be the immediate subtrees of G_X. Let G_0 be the tree obtained by removing G_X from G; i.e., G_0 is the scarred subtree with root at the root of G and with a single scar at X.

During Round 1, the following auxiliary edges are derived: (1) for each ancestor Y of X the edge $Y \to X$ is an auxiliary edge, and (2) for each descendent Y of X the edge $X \to Y$ is an auxiliary edge. By deriving auxiliary edges in the correct order (namely, shorter edges first), only one closure step is needed for each auxiliary edge. There are at most m auxiliary edges and thus fewer than m closure steps are needed in round 1.

The subtrees $G_0, \ldots, G_k$ are disjoint and partition the nodes of G. They will be treated in the next round.

The total number of closure steps used to derive auxiliary edges in Round 2 is less than $\sum m_i < m$.

Round ℓ. The previous round $\ell - 1$ resulted in G being split into multiple, disjoint subtrees of size $\leq m/2^{\ell-1}$. In round ℓ, we separately consider each such subtree H which is of size $m_H \geq 2$ and process it in the manner of round 1. Since the subtree H is of size $m_H \geq 2$, Lemma 2.5 gives a node X in H such that H_X has size $\geq m_H/2$ and each of H_X's immediate subtrees have size $< m_H/2$. Now auxiliary edges are added from each ancestor of X in H to X and from X to each of its descendents in H; there are $\leq m_H$ such auxiliary edges and each can be added with at most one closure step. The immediate subtrees of H_X and the subtree H with H_X removed have size $\leq m_H/2$ and will be treated in next round.

Since the total size of all the disjoint subtrees is no more than m edges, fewer than m closure steps are used in this round.

The process of adding auxiliary edges ends when all the subtrees being considered have size < 2; namely after no more than $\lfloor \log m \rfloor$ rounds. Thus at most $m\lfloor \log m \rfloor$ closure steps are needed for deriving auxiliary edges.

As in the linear case, each of the n closure edges can be obtained with at most a single closure step from the $m\lfloor \log m \rfloor$ auxiliary edges. To prove this, suppose Y and Z are nodes in G and $Y \to Z$ is a closure edge; of course, Y is an ancestor of Z. Find the greatest value of ℓ, such that Y and Z are in the same subtree H considered during round ℓ. Of course, the nodes Y and Z are in different subtrees in the next round. Hence the node X chosen to split subtree H in round ℓ has Y as an ancestor and Z as a descendent (or possibly, $X = Y$ and

Z is a descendent of X). Thus the edges $Y \to X$ and $X \to Z$ are auxiliary edges derived during round ℓ and the closure edge can be added with a single further closure step.

Thus the total number of inference steps needed for a solution of the serial transitive closure problem is less than $n + m\lfloor \log m \rfloor$ and Theorem 2.6 is proved. □

The rest of proof of Theorem 2.3 proceeds by proving the following theorem by induction on i.

THEOREM 2.7. *Let $i \geq 0$. If the directed graph G is a tree then the serial transitive closure problem has a solution with $(1 + 2i)(n + m \log^{(*i)} m)$ steps.*

Proof. When $i = 0$, the theorem is just a restatement of Theorem 2.6. So fix $i \geq 1$ and assume the theorem holds for $i - 1$. We prove the theorem by splitting G into subtrees of size $\log^{(*i-1)} m$, adding auxiliary edges, and using the auxiliary edges to derive some of the closure edges; we iterate this process $\log^{(*i)} m$ many times, after which the subtrees all have size ≤ 1. The derivation of the closure edges from the auxiliary edges will depend on the induction hypothesis.

Let us begin by describing the reduction process, which will be used iteratively. The input to the reduction process is a subtree T of G; we assume T has $M > 1$ edges. The output of the reduction process will consist of a set of node-disjoint subtrees of T and the derivation of the closure edges whose endpoints are in T but are in different subtrees output by T. The reduction process has three steps:

Step 1. In the first step, T is partitioned into subtrees and auxiliary edges are derived.

By iteratively applying Lemma 2.5, T can be split into a finite set of subtrees $T_0, \ldots, T_k$ so that (1) the edges of the T_j's partition the edges of T; (2) for each $j > 0$, T_j has size $\geq \log^{(*i-1)} M$ edges; (3) for all $j \geq 0$, each immediate subtree of T_j has $< \log^{(*i-1)} M$ edges; and (4) T_0 has root at at the root of T and the rest of the T_j's have a root that is a scar of another subtree in the partition. Clearly there will be at most $\lceil M / \log^{(*i-1)} M \rceil$ many subtrees in the partition.

The following auxiliary edges are derived in Step 1: (1) for each T_j with root X, the edges $X \to Z$ for all other nodes Z of T_j are *auxiliary edges of the first kind*; and (2) for each j and p such that the root Y of T_p is a scar of T_j, the edges $X \to Y$ for all ancestors X of Y in T_j are *auxiliary edges of the second kind.* Figure 2 illustrates the choice of auxiliary

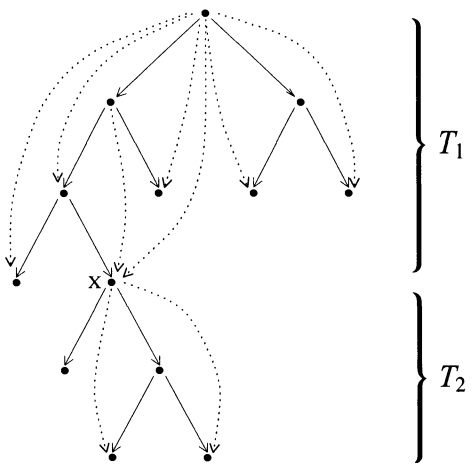


FIG. 2. *Node X is the root of T_2 and a scar of T_1. Hand-drawn edges are the auxiliary edges derived in Step* 1. *The edges of the second kind are the two edges with head X; one of these is also of the first kind.*

edges. It is easy to see that by deriving shorter edges first, each auxiliary edge can be derived by single closure step. Further, we claim that there are $\leq 2M$ auxiliary edges. It is easy to see that there $\leq M$ auxiliary edges of the first kind, since each node in T is at the head of at most one such auxiliary edge. To bound the edges of the second kind, note that if T_j has the root Y of T_p as a scar then the ancestors of Y in T_j consist of the root of T_j and some of the nodes in one of the immediate subtrees of T_j. The edge from the root of T_j to Y is also an edge of the first kind and has already been derived. Hence there are $< \log^{(*i-1)} M$ auxiliary edges from nodes inside T_j to Y. Also, the root of T_p cannot be the root of T (i.e., $p \neq 0$) so there are at most $M/\log^{(*i-1)} M$ different trees T_p to consider. Taking the product of the number of subtrees and the number of edges, we have that there are less than M auxiliary edges of the second kind.

Step 2. In this step we merely describe the output of the reduction process. The *output trees* are precisely the set of immediate subtrees of $T_0, \ldots, T_k$. Note that the output trees are disjoint and partition the nodes of T other than the root node of T, but do not contain all the edges of T. Each output tree has $< \log^{(*i-1)} M$ edges.

Step 3. In the third step we derive every closure edge $X \to Y$ with X and Y in T but in different output trees. Let N be the number of such closure edges. We derive these closure edges by setting up a new instance of the serial transitive closure problem and applying the induction hypothesis. The new instance will consist of a directed graph G', which has as nodes the roots of the trees $T_0, \ldots, T_k$ and has as edges the auxiliary edges from Step 1 that connect these roots. The closure edges of the new instance are the edges $X' \to Y'$, which are obtained by the following method: for each closure edge $X \to Y$ (of the original problem) such that X is a node in T_j and Y is a node in T_p with $j \neq p$, let Y' be the root of T_p and let X' be the (scarred) leaf of T_j such that Y is a descendent of X'. It will be important that $X \to X'$ and $Y' \to Y$ are auxiliary edges (of the second and first kind, respectively).

Clearly the new instance of the serial transitive closure problem has $< M/\log^{(*i-1)} M$ edges in G' and $\leq N$ closure edges. By the induction hypothesis, it has a solution of size less than or equal to

$$(1 + 2(i-1))\left[N + \frac{M}{\log^{(*i-1)} M} \log^{(*i-1)}\left(\frac{M}{\log^{(*i-1)} M}\right)\right],$$

which is trivially bounded by

$$(1 + 2(i-1)) \cdot [N + M].$$

Given a solution to the new serial transitive closure problem, for all X, X', Y, and Y' as above, we can derive the closure edge $X \to Y$ in two closure steps from the auxiliary edges $X \to X'$ and $Y' \to Y$ and the closure edge $X' \to Y'$ of the new problem.

To conclude the description of the reduction process, we note that the total number of closure steps needed in the reduction process is bounded by

$$2M + (1 + 2(i-1))(N + M) + 2N,$$

which is more suggestively written as

$$(1 + 2i)(N + M).$$

The overall procedure for proving Theorem 2.7 can now be very simply explained in terms of iterating the above reduction process.

Round 1. Apply the reduction process to the whole tree G. This derives n_1 closure edges (n_1 is the value of N from the reduction process) and outputs a set of subtrees of G

that partition the nonroot nodes of G and are each of size $< \log^{(*i-1)} m$ edges. The total number of closure steps in Round 1 is bounded by

$$(1 + 2i)(n_1 + m).$$

Round ℓ. The previous round generated a set of node-disjoint subtrees each of size less than

$$\underbrace{\log^{(*i-1)}(\log^{(*i-1)}(\cdots(\log^{(*i-1)}}_{\ell-1 \text{ times}}(m))\cdots)).$$

Apply the reduction process (Steps 1–3) to all of these subtrees that contain more than one edge; the overall result is that some number n_ℓ of closure edges are derived and that a set of node-disjoint output trees each of size less than

$$\underbrace{\log^{(*i-1)}(\log^{(*i-1)}(\cdots(\log^{(*i-1)}}_{\ell \text{ times}}(m))\cdots))$$

is generated. The total number of closure steps in round ℓ is less than

$$(1 + 2i)(n_\ell + m).$$

The rounds are iterated until all the subtrees have size ≤ 1; namely, in no more than $\log^{(*i)} m$ rounds. At the end every closure edge has been derived. The total number of closure steps used is bounded by

$$\sum_{\ell=1}^{\log^{(*i)} m} (1 + 2i)(n_\ell + m)$$

and since $\sum n_\ell = n$, the total number of closure steps is bounded by

$$(1 + 2i)(n + m \log^{(*i)} m).$$

That completes the proofs of Theorems 2.7 and 2.3. □

We are now ready to prove Main Theorem 2.4.

Proof. By Theorem 2.7, the serial transitive closure problem for the graph G has a solution with $O((1 + 2i)(n + m \log^{(*i)} m))$ steps, for any value of i. Let $i = \alpha(m)$: by the definition of the function α, we have $\log^{(*i-1)} m < i$. By Proposition 2.2, it follows that $\log^{(*i)} m \leq 4$. Hence the serial transitive closure problem of G has a solution of size bounded by

$$(1 + 2\alpha(m))(n + 4m) = O((n + m)\alpha(m)). \quad \square$$

We next give a bound on the number of steps in terms of Tarjan's inverse Ackermann function.

COROLLARY 2.8. *If the directed graph G is a tree then the serial transitive closure problem has a solution with $O((n + m)\alpha(n + m, m))$.*

Proof. We argue similarly to the above proof, but now let i equal $\max\{1, \alpha(n+m, m)\}$. Thus, by the definition of $\alpha(,)$, we have $A(i, 4\lceil(n+m)/m\rceil) > \log m$. By Proposition 2.1, $\log^{(*i-1)}(\log m) < 4\lceil(n+m)/m\rceil$, and hence $\log^{(*i)}(\log m) < 4\lceil(n+m)/m\rceil$. Since $i \geq 1$, $\log^{(*i)}(m) \leq 4\lceil(n+m)/m\rceil$. Thus, by Theorem 2.7, the serial transitive closure problem has a solution with size bounded by

$$(1 + 2\alpha(n + m, m))\left(n + m\left(4\left\lceil\frac{n+m}{m}\right\rceil\right)\right) = O((n + m)(\alpha(n + m, m)))$$

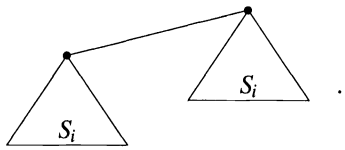
whenever $\alpha(n + m, m) > 0$. □

3. Lower bounds.

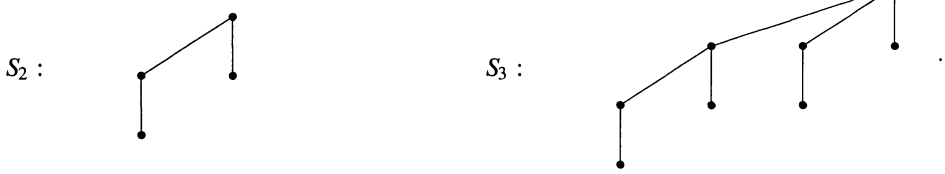
3.1. The lower bound for trees. In this section we prove that our method of solving the serial transitive closure problem for trees is optimal to within a constant factor; in particular, linear size solutions are not always possible. Our proof is based on a theorem of Tarjan [13] giving a lower bound on the worst case runtime of an algorithm for the union-find problem. Tarjan's lower bound applied only to a particular type of algorithm for the union-find problem (Tarjan [14], Fredman and Saks [11], and La Poutré [12] have since given lower bounds for a much wider class of algorithms). Essentially, Tarjan gave a lower bound for algorithms that rely exclusively on manipulating pointers (with certain constraints). Since the serial transitive closure problem is framed so as to deal only with edges and closure steps, it is not so surprising that we are able to modify Tarjan's original construction so as to give a lower bound for the serial transitive closure problem.

Our lower bound is obtained by constructing instances of the serial transitive closure problem that require at least $\frac{1}{25}(n+m)\cdot\alpha(n+m)$ closure steps in any solution. The construction is based on Tarjan's lower bound; we will review the relevant definitions and constructions but do not repeat Tarjan's proof.

Following Tarjan, define S_1 be the tree with two nodes: the root and one leaf. Define S_{i+1} to be the tree constructed by making two copies of S_i and making the root of one copy a child of the root of the other copy; pictorially, S_{i+1} is



For example, S_2 and S_3 are the trees



Clearly S_i has 2^i nodes of which 2^{i-1} are leaves. Let T be a tree. A *g-find* is a pair of nodes (a, b) in T such that b is a leaf and a is an ancestor of b. A (permissible) sequence of g-finds is a sequence (a_i, b_i) of g-finds such that the b_i's are distinct and such that, for all $i < j$, a_j is not a descendent of a_i. G is a *shortcut graph* of T if G is a directed graph on the nodes of T with each edge of T contained as a directed edge in T such that, if T is viewed as a directed graph with edges directed downward, the graph G is a subgraph of the transitive closure of T. If (a_i, b_i), $i = 1, \ldots, N$, is a sequence of g-finds, then the associated shortcut graphs $G_0, \ldots, G_N$ are defined by letting G_0 have the edges from T (directed downwards), and letting G_i be G_{i-1} plus all edges (c, d) such that c and d are on the path in T from a_i to b_i and such that d is a descendent of c. The *cost* of a particular g-find (a_i, b_i) in the sequence is defined to be the distance from a_i to b_i in the shortcut graph G_{i-1}.

Tarjan defines a version of the Ackermann function that is slightly different from ours; however, it is easy to see that Tarjan's Ackermann function equals ours, except for the values of $A(i, 0)$.

The next theorem of Tarjan will lead to our lower bounds.

THEOREM 3.1. *For all $k \geq 1$ and $i > A(4k, 4)$, there is a sequence of 2^{i-2} g-finds in S_i such that each g-find has cost $\geq k$.*

For a proof of Theorem 3.1, apply [13, Thm. 15] with $s = 1$. We can now prove our lower bound.

THEOREM 3.2. *Let $k \geq 1$. There is an instance of the serial transitive closure problem in which the graph G has $m = 2^{A(4k,4)+1}$ nodes and for which there are $n = 2^{A(4k,4)-1}$ closure edges such that any solution of of the serial transitive closure problem requires at least $(k-1) \cdot 2^{A(4k,4)-1}$ closure steps.*

Proof. Let the directed graph G be the tree S_i (with edges directed downward), where $i = A(4k, 4) + 1$. Clearly G has $m = 2^{A(4k,4)+1}$ nodes. The instance of the serial transitive closure problem is obtained by taking as closure edges, the $n = 2^{A(4k,4)-1}$ g-finds given by Theorem 3.1. Let $G_0, \ldots, G_n$ be the associated shortcut graphs.

We now must show that, because each g-find has cost k, any solution to this serial transitive closure problem requires at least $k-1$ steps per closure edge. To prove this, let $e_1, \ldots, e_s$ be a sequence of edges that comprise a solution to the serial transitive closure problem. For each $i \leq s$, define the *rank* of e_i to be the least value r_i such that e_i is an edge in G_{r_i}; let $r_i = n + 1$ if e_i is not in G_n. Now reorder the edges $e_1, \ldots, e_s$ according to their rank, keeping edges of the same rank in the same relative order. We claim that the reordered edges are also a solution to the serial transitive closure problem. This claim is easily proved by noting that if an edge $e_i = (c_1, c_2)$ is inferred from edges (c_1, c_3) and (c_3, c_2) and if e_i is in G_{r_i} then the other two edges must also be in G_{r_i} (by the definition of the shortcut graphs) and thus have ranks $\leq r_i$. Finally, we claim that that for all $r \leq n$ there are at least $k-1$ edges with rank r. To prove this, consider the edges of rank r; these edges are inferred by closure steps from the edges of rank $< r$ and one of the edges is the rth closure edge. But because the cost of the rth g-find is $\geq k$, at least $k-1$ closure steps are required to derive the rth closure edge from the edges of rank $< r$. □

Let $f(m, n)$ be the maximum number of edges required to solve serial transitive closure problems for *trees* with m edges and n closure edges.

LEMMA 3.3. *For infinitely many values of n, $f(4n-1, n) \geq \frac{n}{5}\alpha(5n)$.*

Proof. Let $k \geq 5$ and $n = 2^{A(4k,4)-1}$ and $i = A(4k, 4) + 1$. Since S_i has $4n$ nodes, it has $m = 4n - 1$ edges. By Theorem 3.2, $f(m, n) \geq (k-1)n$. It will suffice to show $\alpha(5n) \leq 5k - 5$. Now,

$$\begin{aligned} A(5k-5, 5k-5) &> A(5k-5, 5) \\ &= A(5k-6, A(5k-5, 4)) \\ &\geq A(5k-6, A(4k, 4)) \\ &> 2^{A(4k,4)+2} \qquad \text{since } 5k-6 > 2 \\ &> 5n \end{aligned}$$

from whence, by the definition of α, $\alpha(5n) \leq 5k - 5$. □

Since $m = 4n - 1$ and hence $5n > m + n$, the proof of Lemma 3.3 immediately implies the following result.

THEOREM 3.4. *$f(m, n) \geq \frac{1}{25}(n + m) \cdot \alpha(n + m)$ for infinitely many values of n with $m = 4n - 1$.*

3.2. Relations to weak superconcentrators. The above lower bound for the serial transitive closure problem applies to the case where G is a general tree. Essentially the same lower bound applies to binary trees, since any tree can be converted into a binary tree by expanding any node that has more than two children into multiple nodes with two children each. This expansion will at most double the number of edges in the tree and will not make the solutions to the serial transitive closure problem smaller.

However, we know of no way to extend the above argument to give a nonlinear lower bound for the size of solutions to the serial transitive closure problem for the case where the directed graph is linear, i.e., the case where each node in G has at most one incoming edge and at most one outgoing edge (this was the simple case considered in the preliminary portion of the proof of Theorem 2.6). On the other hand, there is a connection between constant depth weak superconcentrators and our proof of the main theorems, that suggests that our upper bound of $n \cdot \alpha(n)$ is the best that can be obtained for the linear case with our techniques. Namely, our proof of Theorem 2.7, for the case where G is linear, implicitly contains a construction of weak superconcentrators of optimal size (to within a constant factor). We outline this construction below; a similar construction has already been given by [6], [7].

It is also possible to use Yao's lower bound on the size of (t, m)-structures [15] to get a nonlinear lower bound on the number of auxiliary edges needed for our construction. However, the lower bounded obtained in this way is not as good as the lower bound we obtain below via weak superconcentrators.

We recall the definition of a weak superconcentrator. A network is a directed graph with nodes $a_0, \ldots, a_m$ designated as inputs and with nodes $b_0, \ldots, b_m$ designated as outputs. The inputs have no incoming edges and the outputs have no outgoing edges. A network is *synchronous* if it is possible to associate with each node a *depth*, such that each input has depth 0 and each edge in the network goes from a depth d node to a depth $d+1$ node, and such that the output nodes have a common depth. The depth of a synchronous circuit is defined to be the depth of the output nodes. The network is said to be a *weak* $(m+1)$*-superconcentrator* if the following property holds: if $0 \leq i_1 \leq j_1 < i_2 \leq j_2 < \ldots i_k \leq j_k \leq m$ are integers, then the network contains k many node-disjoint paths from a_{i_r} to b_{j_r}, for $r = 1, \ldots, k$. It is a theorem of Dolev, Dwork, Pippenger and Widgerson [10] that depth $2i + 2$ synchronous weak $(m + 1)$-superconcentrators must have at least $\Omega(m \log^{(*i)}(m))$ many nodes.[1] (Some related, but weaker, lower bounds are given by Bodlaender, Tel, and Santoro [1].)

Recall that in the proof of Theorem 2.7, we added auxiliary edges independently of the choice of closure edges and then derived the closure edges with the aid of the auxiliary edges. The net effect is, after the proof by induction has been unwound, that there were a total of $(1 + 2i)m \log^{(*i)}(m)$ auxiliary edges added such that, for any two nodes A and B with A an ancestor of B, there exists a path from A to B of length at most $(2i + 2)$.[2] This path consists entirely of auxiliary edges and edges in G, of course. We claim that these auxiliary edges can be made into a weak supercentrator of depth $(2i + 2)$. To do this we make $(2i + 3)$ disjoint copies of G, called $G_0, \ldots, G_{2i+2}$ and construct a network on the nodes in the union of these graphs.

To construct the weak $(m + 1)$-superconcentrator of depth $2i + 2$, let G be a linear tree with nodes $X_0, \ldots, X_m$ and edges $X_j \to X_{j+1}$, for $0 \leq j \leq m$. Form $(2i+3)$ disjoint copies of G denoted $G_0, \ldots, G_{2i+2}$ and let X_j^k be the copy of X_j in G_k. The weak superconcentrator will be a directed graph on the nodes X_j^k; the X_j^0's are the input nodes, the X_j^{2i+2}'s are the output nodes and each X_j^k will be of depth k. First, the weak superconcentrator will have the edges $X_j^k \to X_j^{k+1}$ for all j, k. There are obviously $2(i + 1)(m + 1)$ edges of this first kind. Second, the weak superconcentrator will have edges corresponding to the auxiliary edges from the proofs of Theorems 2.6 and 2.7. Recall that Theorem 2.7 was proved by

[1] The Ackermann function $A(i, x)$ used in [10] is the same as Tarjan's Ackermann function, and hence is the same as our $A(i, x)$ for $x \geq 1$. The function $\lambda(i, x)$ used in [10] satisfies $\log^{(*i)}(x) = \lambda(i + 1, x)$ for all $i \geq 0$ and $x > 0$.

[2] To justify the length $(2i + 2)$ recall that at most $(2i + 1)$ closure steps were needed to derive any possible closure edge from the auxiliary edges — this corresponds to a path of length $(2i + 2)$.

induction on i; by considering the auxiliary edges added to G in the proof of Theorem 2.6 and in all the induction steps of the proof of Theorem 2.7, it is easy to see that there are, in total, $\leq (1+2i)m \log^{(*i)} m$ auxiliary edges added to G. We must explain how the auxiliary edges are translated into edges in the weak superconcentrator — the essential idea is that each induction step of the proof of Theorem 2.7 adds two more layers of connections in the weak superconcentrator. To make this more precise, recall that in proving Theorem 2.7 for i we added $2m \log^{(*i)}(m)$ many auxiliary edges and invoked Theorem 2.7 for $i-1$ multiple times. If this proof by induction is "unwound," then each of these invocations of Theorem 2.7 for $i-1$ adds many auxiliary edges and further invokes Theorem 2.7 for $i-2$ multiple times, etc. The proof of Theorem 2.7 for $i=0$ was just the proof of Theorem 2.6 (in particular, the linear case); this of course also added auxilliary edges. Now consider all the auxiliary edges that are added during the unwinding of the proof of Theorem 2.7 for i; each auxiliary edge is associated with a value $i' \leq i$ by considering which case of Theorem 2.7 introduced the auxiliary edge (edges can be associated with more that one value of i' and in this case we count the edge multiple times). The weak superconcentrator contains all edges $X_j^{i-i'} \to X_s^{i-i'+1}$ and $X_j^{i+i'+1} \to X_s^{i+i'+2}$ where $X_j \to X_s$ is an auxiliary edge associated with i'. Since each auxiliary edge is put twice into the weak superconcentrator,[3] there are $2(1+2i)m \log^{(*i)}(m)$ such edges put into the network.

Thus, in total, there are only $O(i \cdot m \cdot \log^{(*i)} m)$ edges in the weak $(m+1)$-superconcentrator of depth $2i+2$. Hence our method of proof for the upper bound of Theorem 2.7 constructs optimal size weak superconcentrators and it seems, therefore, that no simple modification of the proof method can give a better upper bound for the linear case. Nonetheless, it is open whether our upper bound for the linear case of the serial transitive closure problem is optimal — for instance, it might be possible to give an improved construction by not choosing the auxiliary edges independently of the closure edges. The point is that it is not a priori necessary to construct a weak superconcentrator in order to get a solution to a particular instance of the serial transitive closure problem. The lower bound methods of [13] and [10] do not appear to give a definitive lower bound for the linear case.

Acknowledgments. We thank P. Pudlák for suggesting the connection to weak superconcentrators. We also thank one of the referees for bringing the works of Yao, Chazelle, and Bodlaender, Tel, and Santoro to our attention and suggesting some improvements to the paper, including Theorem 2.8.

REFERENCES

[1] H. L. BODLAENDER, G. TEL, AND N. SANTORO, *Trade-Offs in Non-Reversing Diameter*, Tech. Rep. RUU-CS-89-22, Dept. of Computer Science, Utrecht Univ., 1989.

[2] M. L. BONET, *The Lengths of Propositional Proofs and the Deduction Rule*, Ph.D. thesis, Dept. of Mathematics U.C. Berkeley, 1991.

[3] M. L. BONET AND S. R. BUSS, *The deduction rule and linear and near-linear proof simulations*, Journal of Symbolic Logic, 58 (1993), pp. 688–709.

[4] ———, *On the deduction rule and the number of proof lines*, in Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science, 1991, pp. 286–297.

[5] R. P. BRENT, *The parallel evaluation of general arithmetic expressions*, J. Assoc. Comput. Mach., 21 (1974), pp. 201–206.

[6] A. K. CHANDRA, S. FORTUNE, AND R. LIPTON, *Lower bounds for constant depth circuits for prefix problems*, in 10th International Colloquium on Automata, Languages and Progamming, Springer-Verlag Lecture Notes in Computer Sci. Vol. 154, 1983, pp. 109–117.

[3] It is possible to improve the construction to put each auxiliary edge only once in the weak superconcentrator.

[7] A. K. CHANDRA, S. FORTUNE, AND R. LIPTON, *Unbounded fan-in circuits and associative functions*, in Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 1983, pp. 52–60.

[8] B. CHAZELLE, *Computing on a free tree via complexity-preserving mappings*, Algorithmica, (1987), pp. 337–361.

[9] M. D. DAVIS AND E. J. WEYUKER, *Computability, Complexity, and Languages*, Academic Press, 1983.

[10] D. DOLEV, C. DWORK, N. PIPPENGER, AND A. WIDGERSON, *Superconcentrators, generalizers and generalized connectors with limited depth*, in Proceedings 15th Annual ACM Symposium on Theory of Computing, 1983, pp. 42–51.

[11] M. L. FREDMAN AND M. E. SAKS, *The cell probe complexity of dynamic data structures*, in Proceedings 21st Annual ACM Symposium on Theory of Computing, 1989, pp. 345–354.

[12] J. L. POUTRÉ, *Lower bounds for the union-find and the split-find problem on pointer machines*, in Proceedings of the 22th Annual ACM Symposium on Theory of Computing, 1990, pp. 34–44.

[13] R. E. TARJAN, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach., 22 (1975), pp. 215–225.

[14] ———, *A class of algorithms which require nonlinear time to maintain disjoint sets*, J. Comput. System Sci., (1979), pp. 110–127.

[15] A. C. YAO, *Space-time tradeoffs for answering range queries (extended abstract)*, in Proceedings of the 14th Annual ACM Symposium on Theory of Computing, 1982, pp. 128–136.

SIAM J. COMPUT.
Vol. 24, No. 1, pp. 123–137, February 1995

008

NEW DECIDABILITY RESULTS CONCERNING TWO-WAY COUNTER MACHINES*

OSCAR H. IBARRA†, TAO JIANG‡, NICHOLAS TRAN§, AND HUI WANG¶

Abstract. The authors study some decision questions concerning two-way counter machines and obtain the strongest decidable results to date concerning these machines. In particular, it is shown that the emptiness, containment, and equivalence (ECE for short) problems are decidable for two-way counter machines whose counter is reversal-bounded (i.e., the counter alternates between increasing and decreasing modes at most a fixed number of times). This result is used to give a simpler proof of a recent result which shows that the ECE problems for two-way reversal-bounded pushdown automata accepting bounded languages (i.e., subsets of $w_1^* \dots w_k^*$ for some nonnull words $w_1, \dots, w_k$) are decidable. Other applications concern decision questions about simple programs. Finally, it is shown that nondeterministic two-way reversal-bounded multicounter machines are effectively equivalent to finite automata on unary languages, and hence their ECE problems are decidable also.

Key words. counter machines, decidability, emptiness problem, pushdown automata, simple programs

AMS subject classifications. 68Q05, 68Q68, 68Q75

1. Introduction.

A fundamental decision question concerning any class C of language recognizers is whether there exists an algorithm to decide the following question: given an arbitrary machine M in C, is the language accepted by M empty? This is known as the emptiness problem (for C). Decidability (existence of an algorithm) of emptiness can lead to the decidability of other questions such as containment and equivalence (given arbitrary machines M_1 and M_2 in C, is the language accepted by M_1 contained (respectively, equal) to the language accepted by M_2) if the languages defined by C are effectively closed under union and complementation.

The simplest recognizers are the finite automata. It is well known that all the different varieties of finite automata (one-way, two-way, etc.) are effectively equivalent, and the class has decidable emptiness, containment, and equivalence (ECE, for short) problems.

When the two-way finite automaton is augmented with a storage device, such as a counter, a pushdown stack or a Turing machine tape, the ECE problems become undecidable (no algorithms exist). In fact, it follows from a result in [9] that the emptiness problem is undecidable for two-way counter machines even over a unary input alphabet. If one restricts the machines to make only a finite number of turns on the input tape, the ECE problems are still undecidable, even for the case when the input head makes only one turn [8]. However, for one-way counter machines, it is known that the equivalence (hence also the emptiness) problem is decidable, but the containment problem is undecidable [11]. The situation is different when we restrict both the input and counter. It has been shown that the ECE problems are decidable for counter machines with a finite-turn input and a reversal-bounded counter (the number of alternation between increasing and decreasing modes is finite, independent of the input) [8]. It is also known that the ECE problems are decidable for two-way counter machines with a reversal-bounded counter accepting bounded languages (subsets of $w_1^* \dots w_k^*$ for some nonnull words $w_1, \dots, w_k$) [6]. Note that these machines can accept fairly complex languages. For example,

*Received by the editors November 19, 1992; accepted for publication (in revised form) September 23, 1993.

†Department of Computer Science, University of California, Santa Barbara, California 93106. The research of this author was supported in part by National Science Foundation grant CCR89-18409.

‡Department of Computer Science and Systems, McMaster University, Hamilton, Ontario L8S 4K1, Canada. The research of this author was supported in part by Natural Sciences and Engineering Research Council of Canada operating grant OGP 0046613.

§Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104.

¶Department of Computer Science, University of Alabama, Huntsville, Alabama 35899.

it can recognize the language consisting of strings of the form 0^i1^j where i divides j. The decidability for the general case when the input is not over a bounded language was left open in [6]. We resolve this question here: We show that the ECE problems are decidable for two-way reversal-bounded counter machines. We believe this is the largest known class of machines, which are a natural generalization of the two-way finite automaton, for which the ECE problems are decidable. This result has some nice applications. We use it to give a simpler proof of a recent result that the ECE problems for two-way reversal-bounded pushdown automata accepting bounded languages (i.e., subsets of $w_1^* \dots w_k^*$ for some nonnull words $w_1, \dots, w_k$) are decidable. Other applications of this result concern decision questions about some special classes of simple programs.

Finally, we consider the nondeterministic version of reversal-bounded multicounter machines. We show that when the input alphabet is unary, these machines accept only regular languages. Since the proof is constructive, we obtain as corollaries that the ECE problems for these machines are also decidable. This resolves an open question raised in [5], where a similar result was shown for deterministic machines. This is the strongest result one can obtain since it is known that deterministic reversal-bounded 2-counter machines over $a_1^* \dots a_k^*$ (for distinct symbols $a_1, \dots, a_k$ for some k) can accept nonsemilinear sets and have an undecidable emptiness problem [8].

The rest of this paper is organized as follows. In §2, we show the decidability of the ECE problems for two-way reversal-bounded counter machines. We give the applications in §3. Finally, we show the effective equivalence of two-way reversal-bounded nondeterministic multicounter machines and finite automata over unary languages, along with the decidability of their ECE problems in §4.

In the remainder of this section, we define the models of computation of interest and related concepts. A *language* L is a subset of $\{0, 1\}^*$. A language is *strictly bounded* over k letters $a_1, a_2, \dots, a_k$ if it is a subset of $a_1^* a_2^* \dots a_k^*$. A language is *bounded* over k nonnull words $w_1, w_2, \dots, w_k$ if it is a subset of $w_1^* w_2^* \dots w_k^*$. A *counter machine* is a finite automaton augmented with a counter, whose value can be incremented, decremented, or tested for zero. An r-input reversal 2DCM is a two-way counter machine whose input head makes at most r reversals (but the counter is unrestricted). A two-way machine is *sweeping* if the input head reverses only on the endmarkers. 2DCM(c, r) denotes the class of deterministic machines having a two-way input head and c counters, each of which makes at most r reversals in any computation. 2NCM(c, r) denotes the corresponding nondeterministic class. 2DPDA(r) denotes the class of two-way deterministic pushdown automata whose pushdown stack makes at most r reversals. 2NPDA(r) denotes the nondeterministic class.

In the following sections, we will study the emptiness problems for the above machines on bounded languages. A straightforward argument shows that a machine of any type studied in this paper accepts a nonempty bounded language if and only if there is another machine of the same type that accepts a nonempty *strictly* bounded language. (Suppose a 2DCM(c, r) M accepts a nonempty bounded language $L \subseteq w_1^* w_2^* \dots w_k^*$. Let h be a homomorphism that maps each word w_i to a new letter s_i for $1 \leq i \leq k$. Then there is another 2DCM(c, r) M' that simulates M on w_i whenever it reads the letter s_i. Clearly, M' accepts $h(L)$.) So when we are dealing with the emptiness question for machines over bounded languages, we need only handle the case when the machines accept strictly bounded languages.

2. Reversal-bounded counter machines. It was shown in [8] that many decision problems such as emptiness, containment, and equivalence for the class 2DCM(c, r) for $c \geq 2$ and $r \geq 1$ are undecidable. The same paper raised the question of the decidability of these problems for 2DCM$(1, r)$ for $r \geq 1$. A partial answer was given in [6], where it was shown that the emptiness problem for 2DCM$(1, r)$ for $r \geq 1$ over bounded languages is decidable.

In this section we give a complete answer to this question; we show that the ECE problems for the class 2DCM(1, m), where $m \geq 1$, are decidable. First, we consider the emptiness problem and improve the result of Gurari and Ibarra by removing the requirement that the language be bounded.

THEOREM 2.1. *The emptiness problem for* 2DCM(1, r) *is decidable for every* $r \geq 1$.

Proof. Fix an $r \geq 1$, and let M be a 2DCM(1, r) with q states. We show that if $L(M) \neq \emptyset$, then M must accept an input in some bounded language over k words $w_1, w_2, \ldots, w_k$, where k and $w_1, w_2, \ldots, w_k$ depend only on M. Suppose M accepts some input x. Without loss of generality, we assume there are exactly $r + 1$ phases in the computation of M on x, such that in each phase the counter is either increasing or decreasing.

Define the *phase crossing sequence* C_i at the boundary of two input squares to be the sequence (s, d) of M's states and input head directions at the times the input head crosses this boundary during phase i. During an increasing phase, the input head crosses any boundary at most $2q$ times, or else M, being deterministic, would get into an infinite loop. Hence, there are at most $(2q)!$ different crossing sequences for an increasing phase. During a decreasing phase, the input head is allowed to go into a loop on the input tape (i.e., crosses some boundary, say b_0, twice in the same state and direction) until the counter becomes empty. In this case, the input head crosses each boundary in the loop $O(j)$ times, where j is the number of loop iterations. (A boundary is *in* the loop if it is crossed by the input head between two crossings of b_0 in the same state and direction.) Each boundary outside the loop is crossed at most a constant number of times.

Thus any crossing sequence during this phase can be written in the form $C_i = uv^j w$, where

1. no pairs of state and direction appear twice in u or v,
2. no pairs of state and direction appear in u *and* in v,
3. w is a prefix of v,
4. $|u| + |v| \leq 2q$.

Hence there are at most $(2q)! \cdot (2q)^2 \leq (2q + 2)!$ different crossing sequences during a decreasing phase. (Note that j is a fixed number for each phase.)

Define the *crossing sequence* at the boundary of two input squares to be the string $C_1\#C_2\#\cdots\#C_{r+1}$ of the phase crossing sequences at that boundary. From the above paragraph, we can see that the number of different crossing sequences is bounded by $c = [(2q + 2)!]^{r+1}$, which is independent of x. Hence we can rewrite x as $u_0v_1u_1 \ldots v_ku_k$, such that $0 \leq |u_j|, |v_j| \leq c$ for $j = 1, \ldots, k$ and the crossing sequences at the left and right boundaries of each v_j are identical. Call each v_j a *movable segment* and each u_j a *separator segment*. We say two movable segments v_m and v_n are of the same type if they are identical and have the same crossing sequences at the left and right boundaries. There are at most ct^{c+1} different types, where t is the size of the input alphabet.

We now transform x to another string x' by performing the following operation repeatedly on x. If $x = z_1v_1z_2v_2z_3$, where v_1 and v_2 are movable segments of the same type, then we rewrite x as $x' = z_1v_1v_2z_2z_3$, i.e., we group two movable segments of the same type together. If M were a DFA, it would be immediate that M accepts x iff it accepts x', because the crossing sequences of v_1 and v_2 are identical. However, the transposition may affect the counter value and alter the times and places of the counter reversals, so that the computations of M on x' may be completely different from the computation of M on x.

Fortunately, this situation cannot happen. In fact, each phase in the computation of M on x cannot end while the input head is reading $v_1z_2v_2$, since the crossing sequences of v_1 and v_2 are identical. This implies that transposing z_2 and v_2 does not affect the times and places of the counter reversals, nor does it affect the total net change to the counter value in each phase by the input segment $v_1z_2v_2$.

Hence, although the computations may not match step by step, each phase in the computation of M on x' ends with the same input head position (on z_1 or z_3), the same counter value, and the same state as with the corresponding phase of M on x. Thus, M accepts x iff M accepts x'.

Apply the same process of partitioning and transforming recursively on z_1 and z_2z_3 (at most $|x|$ times), so that at the end, x can be written as a bounded word $y \in w_1^*w_2^* \dots w_k^*$ where $|w_i| \leq c$, $k \leq ct^{c+1}$, c (hence k) is independent of x, and M accepts x iff M accepts y. But since the emptiness problem for 2DCM$(1, r)$ on bounded languages is decidable [6], the theorem follows. □

In contrast to Theorem 2.1 we state and briefly sketch the proof of the following result from [8].

THEOREM 2.2 [8]. *The emptiness problem for* 1*-input reversal* 2DCM *is undecidable.*

Proof. We reduce the emptiness problem for Turing machines to this problem. Given an arbitrary Turing machine M, we can find two 1DCM, M_1 and M_2, and a homomorphism g_1 such that $L(M) = g(L(M_1) \cap L(M_2))$. Furthermore, we can find a 1NCM M_3 such that $L(M_3) = L(M_2)^R$. By expanding the input alphabet with markers to dictate the nondeterministic choices of M_3, we can obtain a 1DCM M_4 and a homomorphism g_2 (which maps the markers to the null string and leaves the other symbols the same) such that $L(M_3) = g_2(L(M_4))$. We also modify M_1 to include markers in its alphabet; the resulting machine M_5 behaves like M_1 and ignores markers in its input.

From these constructions we have $L(M) = g_1g_2(L(M_5) \cap L(M_4)^R)$, and it is now trivial to construct from M_4 and M_5 a 1-input reversal 2DCM M' such that $L(M) = g_1g_2(L(M'))$. Since the emptiness problem for Turing machines is undecidable, so is the emptiness problem for 1-input reversal 2DCM. □

Next, we show the decidability the equivalence and containment problems. For every machine 2DCM$(1, r)$, we can construct an equivalent halting one. Below we prove that this is in fact true for every 2DCM(c, r). We first prove a lemma which shows how to modify a 2DFA to make it halt on every input. This lemma is slightly stronger than a similar result by Sipser [10], since our simulating 2DFA behaves exactly as the original machine until it detects that looping has occurred.

LEMMA 2.3. *For every* 2DFA N *we can construct an equivalent halting* 2DFA N' *such that on each input* N' *moves exactly like* N *until* N *enters a loop.*

Proof. Let $x = a_1a_2 \dots a_n$ be an input. For simplicity, assume N moves every step. For each i, $1 \leq i \leq n$, define a relation $R(i)$ as follows: For any two states p and q of N, the pair (p, q) is in $R(i)$ iff in state p, N will leave cell i to the right and return to cell i in state q the next time. Clearly N doesn't halt iff it moves to some cell i in state q and (q, q) is in the transitive closure of $R(i)$. We construct a 2DFA N' which simulates N and at the same time tries to construct the relation $R(i)$ for each cell being visited. To simplify the presentation, here we will allow N' to construct the relation $R(i)$ nondeterministically. The nondeterminism can be eliminated by the standard subset construction technique. Let δ_N be the transition function of N, and suppose N starts at cell 1.

N' initializes $R(1)$ to $\emptyset$. In general, suppose that N is at cell j. If N moves to the left, N' constructs deterministically $R(j-1)$ from the current $R(j)$ as follows. For each pair (p, q), N' puts (p, q) in $R(j-1)$ iff

1. $\exists s[\delta_N(p, a_{j-1}) = (s, right)\ \&\ \delta(s, a_j) = (q, left)]$, or
2. $\exists s_1, \dots, s_k[\delta_N(p, a_{j-1}) = (s_1, right), \delta_N(s_k, a_j) = (q, left)\ \&\ (s_1, s_2), \dots, (s_{k-1}, s_k) \in R(j)]$.

If N moves to the right, N' constructs (nondeterministically) $R(j+1)$ from the current $R(j)$ as follows: For each pair (p, q) in $R(j)$, M' guesses a sequence of distinct states

$s_1, \ldots, s_k$, such that $\delta_N(p, a_j) = (s_1, right)$ and $\delta_N(s_k, a_{j+1}) = (q, left)$, and puts all $(s_1, s_2), \ldots, (s_{k-1}, s_k)$ in $R(j+1)$. In doing this, N' also makes sure that no two pairs in the resulting $R(j+1)$ share the same first state. Moreover, N' also checks the validity of the current $R(j)$ and abandons the computation if some pair (p, q) in $R(j)$ is nonrealizable, i.e., $\delta_N(p, a_j) = (p', left)$ for some state p' or there exists no state q' such that $\delta_N(q', a_{j+1}) = (q, left)$.

N' halts if it detects that at some cell j, N is in state q and (q, q) is in current $R(j)$. Clearly, during the simulation, a relation $R(j)$ may contain some nonrealizable pairs. But it is not hard to see that these nonrealizable pairs will not cause a false loop detection. Thus we can establish the following claims. Suppose that N is at cell j.

1. If N is in state q and (q, q) is in the current $R(j)$, then N is in a loop.
2. Suppose that the rightmost cell that N has visited so far is cell k. Then for any pair (p, q) such that in state p, M will leave cell j to the right, stay in the cells $j+1$ through k, and return to cell j for the first time in state q, (p, q) is contained in $R(j)$.

Hence, N' can detect if N will enter a loop correctly. □

THEOREM 2.4. *For any c and $r \geq 1$, we can effective convert a* 2DCM(c, r) *to a* 2DCM(c, r) *that halts on every input.*

Proof. For simplicity, we prove the theorem for the case $c = 1$. The idea is the same for $c > 1$. Let M be a 2DCM$(1, r)$ machine. We construct an equivalent halting 2DCM$(1, r)$ machine M'. M' basically simulates M faithfully. We know that M can enter a loop only when (i) M is in an increasing phase or (ii) M is in an decreasing phase but the moves do not affect (i.e., really decrease) the counter. Call the latter a zero-decrease period. So besides simulating M, M' also checks if M will enter a loop in each increasing phase and each zero-decrease period. The fact that M' is able to realize this follows from the observation that M behaves like a 2DFA in an increasing phase and a zero-decrease period, and the above lemma. □

COROLLARY 2.5. *For each c, $\bigcup_r$* 2DCM(c, r) *is effectively closed under complementation, intersection, and union.*

COROLLARY 2.6. *The containment and equivalence problems for $\bigcup_r$* 2DCM$(1, r)$ *are decidable.*

Open Question: Is 2DCM(c, r) closed under union or intersection?

A related interesting halting result is given below, where 2DCM$(1, \infty)$ denotes a two-way deterministic 1-counter machine whose counter can make unrestricted (unbounded) number of reversals.

THEOREM 2.7. *Each* 2DCM$(1, \infty)$ *machine accepting a bounded language can be made halting.*

Proof. We first prove the theorem for unary languages. Let M be a 2DCM$(1, \infty)$ machine that accepts $L \subset a_1^*$. We construct a 2DCM$(1, \infty)$ M' that halts on all inputs and accepts the same language. We may assume that upon acceptance M halts in a unique accepting configuration.

First, observe that if M halts on some input x, then during the computation, the value of the counter cannot exceed $(q+1)|x|$, where q is the number of states of M. Otherwise, let t be the first time the counter value of M on input x reaches $(q+1)|x|$. For each value $v < (q+1)|x|$, let (q_v, i_v) be the state and the head position of M the last time the counter value reaches v before time t. Since there are at most $q|x|$ different combinations, there are two values v and w such that $q_v = q_w$ and $i_v = i_w$. But this implies that M will get into an infinite loop on x, contradicting the assumption that M halts on x. Second, we may assume M is normalized so that the input head moves from one endmarker to the other without reversing the direction in between.

M' starts with the unique accepting configuration of M and performs a Sipser search for the initial configuration. (A Sipser search performs a depth-first search on the connected component (actually a tree) that contains the unique accepting configuration of the computation graph of M on x. The vertices of the computation graph are the configurations, and the edges reflect the transition function of M. It was shown [10] that such a search does not use more space than originally required by M for space-bounded Turing machines. Here we adapt the technique to work for counter machines.)

During the search, whenever the input head of M' reaches an endmarker, then M' verifies that the counter value is at most $(q + 1)|x|$. If this is the case, M' continues the search; otherwise, M' abandons the current path of the search tree and continues with the next one. Because M' makes sure that the counter value is within bound, the number of nodes in the search tree is finite, and hence the Sipser search is guaranteed to halt. M' accepts iff the search fails. It is clear that M' accepts L and M' always halts.

The proof can be trivially extended to hold for strictly bounded languages over k letters for any $k > 1$. The simulating machine M' now performs the check whenever it reaches a boundary between the k segments of the input, instead of the endmarkers. A more involved argument is needed to show that the theorem holds for general bounded languages, since now there are many ways to partition the input into repetitions of the k basic words. M' needs to maintain in its finite control all possible partitions of the input in order to resume the computation after performing the check. □

We obtain from the above theorem the next corollary.

COROLLARY 2.8. *The class of* 2DCM(1, ∞) *on bounded languages is effectively closed under complementation, intersection, and union.*

Open Question: Is 2DCM(1, ∞) closed under complementation or union?

3. Some applications. We give three applications of Theorem 2.1 in this section. First we give a simpler proof of the decidability of the ECE problems concerning reversal-bounded deterministic pushdown automata on bounded languages. This result first appeared in [7] as a corollary of a rather difficult theorem.

THEOREM 3.1. *The emptiness problem for* 2DPDA(r) *for* $r \geq 1$ *on bounded languages is decidable.*

The proof follows from Theorem 2.1 and the following lemma.

LEMMA 3.2. *Let M be a* 2DPDA(r) *accepting a strictly bounded language over $a_1^* \dots a_k^*$. We can effectively construct a* 2DCM(1, r) *machine M' (not necessarily accepting the same language) such that $L(M)$ is empty iff $L(M')$ is empty.*

Proof. We may assume, without loss of generality, that on every step M pushes exactly one symbol on top of the stack, does not change the top of the stack, or pops exactly one symbol, i.e., M is not allowed to rewrite the top of the stack. In the discussion that follows, we assume M is processing an input that is accepted, i.e., the computation is halting.

A writing phase is a sequence of steps which starts with a push and the stack is never popped (i.e., the stack height does not decrease) during the sequence. A writing phase is periodic if there are strings u, v, w with v nonnull such that for the entire writing phase, the string written on the stack is of the form uv^iw for some i (the multiplicity) , and the configuration of M (state, symbol, top of the stack) just before the first symbol of the first v is written is the same as the configuration just before the first symbol of the second v is written. Note that w is a prefix of v. Clearly, a writing phase can only end when the input head reaches an endmarker or a boundary between the a_i's. A writing phase can only be followed by a popping of the stack (i.e., reversal) or another writing phase with possibly different triple (u, v, w).

By enlarging the state set, we can easily modify M so that all writing phases are periodic. One can easily verify that because M is reversal-bounded, there are at most a fixed number t of writing phases (in the computation), and t is effectively computable from the specification of M.

We now describe the construction of M'. Let $x = a_1^{i_1} \ldots a_k^{i_k}$ be the input to M. The input to M' is of the form: $y\#c_1\#c_2\# \ldots \#c_t$, where the c_i's are unary strings; y is x but certain positions are marked with markers $m_1, m_2, \ldots, m_t$. Note that a position of y can have 0, 1, ... at most t markers.

M' simulates M on the segment y ignoring the markers. The c_i's are used to remember the counter values. Informally, every time the counter enters a new writing phase, the machine "records" the current value of the counter by checking the c_i's.

M' begins by simulating M on y (ignoring the markers). When a writing phase is entered, M' records the triple (u_1, v_1, w_1) in its finite control and the multiplicity in the counter. Suppose M enters another writing phase. Then M' checks that the input head is on a symbol marked by marker m_1; hence M' can "remember" the input head position. (If it's not marked m_1, M' rejects.) Then it "records" the current value of the counter on the input by checking that the current value is equal to c_1. (If it's not, M' rejects.) M' restores the input head to the position marked m_1 and resets the counter to 0. It can then proceed with the simulation. Next time M' has to record the value of the counter, M' use the input marker m_2 and checks c_2, while storing the triple (u_2, v_2, w_2) in its finite control, etc. Popping of the stack is easily simulated using the appropriate triple (u, v, w) and the counter value. Note that if in the simulation of a sequence of pops, the counter becomes 0, M' must first retrieve the appropriate c_i (corresponding to the pushdown segment directly below the one that was just consumed) and restore it in the counter before it can continue with the simulation; retrieving and restoring the count in the counter requires the input head to leave the input position, but M' can remember the "new" position with a new "marker." If after a sequence of pops M enters a new writing phase before the counter becomes 0, the "residual" counter value is recorded as a new c_i like before. We leave the details to the reader. It is clear that M' in in 2DCM(1, r) for some r. □

Since the proof of Theorem 2.4 can be trivially modified to show that reversal-bounded 2DPDAs can be made halting, the class of languages they defined is effectively closed under complementation. Hence, we have the next corollary.

COROLLARY 3.3. *The containment and equivalence problems for* 2DPDA(r) *for* $r \geq 1$ *on bounded languages are decidable.*

As a second application, we use Theorem 2.1 to show the decidability of the emptiness problem for some classes of simple programs. The motivation for the following definition comes from the study of real-time verification [1].

DEFINITION 3.4. *A* simple program *P is a triple (V, X, I), where $V = \{A_1, A_2, \ldots, A_n\}$ is a finite set of* input variables, *$X \notin V$ is the* accumulator, *and $I = (i_1; i_2; \ldots; i_l)$ is a finite list of* instructions *of the following form*:

1. *label s:* $X \leftarrow X + A_i$
2. *label s:* $X \leftarrow X - A_i$
3. *label s: if* $X = 0$ *goto label t*
4. *label s: if* $X > 0$ *goto label t*
5. *label s: if* $X < 0$ *goto label t*
6. *label s: goto label t*
7. *label s: halt*

Note that the program is not allowed to change the value of an input variable.

DEFINITION 3.5. *For a program P, $EMPTY(P)$ = yes if there are integers (positive, negative, or zero) $a_1, a_2, \ldots, a_n$ such that P on this input halts; otherwise $EMPTY(P)$ = no. The emptiness problem for simple programs is deciding given P if $EMPTY(P)$ is yes.*

At present, we do not know if the emptiness problem for simple programs is decidable. However, for some special cases, we are able to show that the problem is decidable.

One such special case is a program whose accumulator crosses the 0 axis (alternates between positive values and negative values) at most k times for some positive integer k independent of the input. Call such a program *k-crossing*, and in general, a program *finite-crossing* if it is k-crossing for some $k \geq 1$.

For example, a program with three inputs A, B, C that checks the relation $A = (B - C)i$ for some i can operate as follows: Add A to the accumulator, and iterate adding B and subtracting C and checking if the accumulator X is zero after every iteration. (The program halts if X is zero and goes into an infinite loop if X is negative.) Although the accumulator alternates between increasing and decreasing modes arbitrarily many times (which depends on the input), the accumulator crosses the 0 axis at most once. So such a program is 1-crossing.

Although finite-crossing programs are quite powerful, there are programs that are not finite-crossing. Here is an example of a program whose accumulator alternates between positive and negative values an unbounded number of times:

```
    X = 0
a:  X = X + A
    if X = 0 goto c
    if X > 0 goto b
    goto d
b:  X = X − B
    if X = 0 goto c
    if X < 0 goto a
d:  goto e
e:  goto d
c:  halt
```

If $B = A + \sqrt{A}$, then there can be $\sqrt{A}$ alternations of the accumulator between positive and negative values.

Again, we apply Theorem 2.1 to show the emptiness problem for finite-crossing simple programs is decidable.

THEOREM 3.6. *The emptiness problem for finite-crossing simple programs is decidable, even when instructions of the form* $X \leftarrow X + 1$ *and* $X \leftarrow X - 1$, *are allowed.*

Proof. We give an algorithm to decide whether a finite-crossing simple program P halts on any input. Suppose P has l instructions and n input variables $A_1, A_2, \ldots, A_n$. Define the instantaneous description $(c_t, \text{sign}(X_t))$ of P at time t to be the label of the instruction being executed and the sign of the accumulator, which is either negative or nonnegative.

Consider a halting computation of P on some set of input values $a_1, a_2, \ldots, a_n$ as given by a sequence of IDs of P from start to finish. Since the accumulator of P alternates between positive and negative values at most k times for some $k \geq 1$, this computation can be divided into at most $k + 1$ phases such that every ID in a phase has the same second component. Because P is deterministic, some ID must be repeated after the first l steps in each phase, and hence the sequence of IDs in each phase can be written in the form of $uv^i w$, where each u, v, and w is a concatenation of at most l IDs.

Using this observation about finite-crossing simple programs, we construct a reversal-bounded counter machine M such that M accepts some input if and only if P halts on some input. Since M can make only a finite number of reversals on its counter, it cannot simulate P faithfully. Rather, M uses the "padding" technique described in Lemma 3.2. For each phase in

the computation of P, M precomputes the effect made on the counter by each possible loop v of instructions and "stores" it in the input, i.e., M verifies that its input is padded with this extra information. From then on, to simulate a loop M needs only add its corresponding net effect to the counter, and hence to simulate a phase, M needs at most a constant number of counter reversals. Also, to simulate the increment and decrement instructions, M introduces two new variables, a_+ and a_-, which hold $+1$ and -1, respectively. We give a precise description of the construction below.

M only accept inputs of the form $0^{a_1}\#0^{a_2}\#\ldots\#0^{a_n}\#0^{a_+}\#0^{a_-}\#0^{b_1}\#0^{b_2}\#\ldots\#0^{b_m}$, where $m = \sum_{i=1}^{l} i!$. M considers the first $n+2$ integers coded in its input to be the input values to P. The last m integers represent the net changes made to the accumulator by all possible loops of instructions of length at most l.

At the start of the computation, M verifies that its input has the desired form, and that the last m integers indeed represent the respective changes to the accumulator by a loop of instructions of length at most l, assuming the first n integers are the input values to P. Once this is verified, M proceeds to simulate P. Because the computation in each phase is periodic, M does not simulate P step by step once P gets into a loop of instruction. Instead, M records the loop in its finite control and then uses the corresponding value in the latter part of the input to make changes directly to the counter.

It is easy to see that M accepts some input iff P halts on some input. Furthermore, M makes at most $2l$ counter reversals in each phase and at most $2lm$ counter reversal in the initial verification process. Hence M makes at most $2l(k+1+m)$ counter reversals during the entire computation, and by Theorem 2.1 it is decidable whether $L(M) = \emptyset$. This proves the theorem. □

Remark. Allowing the use of "constant" instructions of the form $X \leftarrow X+1$ and $X \leftarrow X-1$ makes simple programs computationally more powerful. For example, the relation $R = \{A : 2|A\}$ can easily be verified by a finite-crossing program with constant instructions, but not by any program without constant instructions, even if it is not finite-crossing.

Although finite-crossing programs can check fairly complicated relations, they cannot verify multiplication and squaring.

Define MULT $= \{(A, B, C)|C = A * B\}$ and SQUARE $= \{(A, B)|B = A^2\}$.

CLAIM 3.7. MULT *and* SQUARE *cannot be verified by finite-crossing programs.*

Proof. We use the proof technique in [6]. It is known that there is a fixed polynomial $p(y, x_1, \ldots, x_t)$ with integer coefficients such that it is undecidable to determine for an arbitrary nonnegative integer m whether or not $p(m, x_1, \ldots, x_t)$ has a nonnegative integer solution in $x_1, \ldots, x_t$ [4].

Clearly, finite-crossing programs can do addition and subtraction. If MULT can be verified by a finite-crossing program, then we can effectively construct, for each m, a finite-crossing program P_m over $A_1, \ldots, A_t, B_1, \ldots, B_s$ (for some s) such that P_m halts on $A_1, \ldots, A_t$, $sB_1, \ldots, B_s$ for some $B_1, \ldots, B_s$ if and only if $p(m, A_1, \ldots, A_t) = 0$. P_m uses the programs for MULT, addition and subtraction. Since we can decide emptiness for finite-crossing programs, we can decide if the polynomial has a solution, which is a contradiction.

If SQUARE can be verified, then the relation $R = \{(A, B, C, D, E, F, G, H)|B = A^2, D = C^2, E = A+B, F = E^2, G = F-E, H = A*B\}$ can also be verified by a finite-crossing program. Thus MULT can effectively be verified. But we already know this is impossible. □

Although finite-crossing simple programs may seem equivalent at first glance to counter machines whose counters can become zero only a finite number of times regardless of its input (we call those *finite-reset* counter machines), the latter are in fact much more powerful computational devices. For example, there is a finite-reset counter machine M that accepts a

set S similar to MULT, namely, $S = \{0^x\#(0^y\#)^x : x, y \geq 1\}$. M first checks that its input is of the form $0^x\#0^{y_1}\#0^{y_2}\#\ldots\#0^{y_n}$ and that $n = x$. This requires 1 reset of the counter. Next M verifies that $y_1 \geq y_2$ by adding $1 + y_1 - y_2$ to its counter; M rejects if the counter resets to zero during this operation. Otherwise, $y_1 \geq y_2$ and M adds $2y_2 - y_1$ to the counter so that its value now becomes $1 + y_2$. Note that the counter does not reset during this operation. M now checks that $y_2 \geq y_3$ and so on, until it has verified that $y_1 \geq y_2 \geq \cdots \geq y_n$. Finally, M verifies that $y_1 = y_n$. It is easy to see that the whole computation requires only two resets of the counter.

It follows from the above remark that the emptiness problem for finite-reset counter machines is undecidable, because they can multiply and hence compute the value of any polynomial $p(y, x_1, x_2, \ldots, x_n)$ (see [3] for the proof of an analogous result). This contrasts with Theorem 3.6.

We can use the same technique in Theorem 3.6 to show that the emptiness problem is decidable for finite-reset counter machines on *strictly bounded languages* (and hence bounded languages also; see the remark at the bottom of §1). Each accepting computation of such a machine M can be divided into a finite number of phases at the times the counter value becomes zero. Since the language is strictly bounded, it is easy to see that the counter makes a reversal only at the boundary points, i.e., the endmarkers or the first symbols of each type. The computation in each phase then can be rewritten in the form uv^iw, such that the lengths of u, v, and w are bounded by some constant depending only M, and v starts and ends in the same state and on the same boundary point. Again, the number of different such v's is bounded by a constant. So we can construct a finite-reversal counter machine M', whose input is padded with the net change to the counter value by each possible loop v, such that $L(M)$ is empty iff $L(M')$ is. From this and Corollary 2.8, we have the following.

THEOREM 3.8. *The ECE problems for finite-reset counter machines over strictly bounded languages are decidable.*

It is interesting to note that there are strictly bounded languages that can be recognized by a finite-reset counter machine but not by any finite-reversal counter machine. One example is $L = \{0^a1^b2^c : a = n(b - c) \text{ for some } n \geq 0\}$.

There is a restricted class of simple programs allowing "nondeterminism" that arises in the theory of real-time verification [1], as seen in the following definition.

DEFINITION 3.9. *A* restricted nondeterministic simple program *is a simple program that allows more than one choice of instruction for each label and has the property that the accumulator X is nonnegative (nonpositive) after each instruction of the form $X \leftarrow X + A_i$ ($X \leftarrow X - A_i$).*

It is an open question whether the emptiness problem for this class of simple programs is decidable. However, we can show that the emptiness problem is decidable for restricted *deterministic* simple programs.

THEOREM 3.10. *The emptiness problem for restricted deterministic simple programs is decidable.*

Proof. We give an algorithm to decide whether a restricted deterministic simple program P halts on any input. Suppose P has l instructions and n input variables $A_1, A_2, \ldots, A_n$. Define the instantaneous description $(c_t, \text{sign}(X_t))$ of P at time t to be the label of the instruction being executed and the sign of the accumulator, which can be either positive, negative, or zero.

Consider a halting computation of P on some set of input values $a_1, a_2, \ldots, a_n$ as given by the sequence of IDs of P from start to finish. There can be at most l IDs with accumulator value zero, or else P would never halt. These IDs divide the computation into a finite number of phases. Because P is deterministic, some ID must be repeated after the first $2l$ steps in each phase, and furthermore, no $l + 1$ consecutive IDs in each phase can have the same first

component due to the constraint on the accumulator. Hence the sequence of IDs in each phase can be written in the form $uv^i w$, where each u, v, and w is a concatenation of at most $2l$ IDs.

Using this observation about restricted simple programs, we construct a reversal-bounded counter machine M such that M accepts some input if and only if P halts on some input, using the same technique of Theorem 3.6. We leave the details to the reader. □

We can allow comparisons between input variables and the emptiness problem still remains decidable.

COROLLARY 3.11. *The emptiness problem is decidable for finite-crossing simple programs and restricted deterministic simple programs which use additional instructions of the form*

if $p(A_i, A_j)$ **goto** *label,*

where A_i and A_j are input variables, and the predicate p is $|$ (for divides), $>$, $<$, or $=$.

Proof. Suppose P_1 is a finite-crossing simple program. We construct another program P_2 which precomputes all possible relations between the variables. P_2 consists of many segments, each of which simulates P_1 for the appropriate relations among the variables. After the preprocessing phase, P_2 jumps to the appropriate segment.

In the case of a restricted deterministic simple program P_1, the construction is similar, but because of the constraint on the accumulator, we need to use additional input variables to accomplish the testing of the relations. For example, suppose A and B are input variables, and we need to check whether $A|B$. In this case, we use an additional variable C storing the value $A + B$ which P_2 has to verify at the beginning. Then, P_2 executes the following code segment:

$X = B$
10: $X = X - C$
$X = X + B$
if $X = 0$ **goto** 20
if $X < 0$ **goto** 30
goto 10
20: A divides B
30: A does not divide B

X initially has the value $A - B$. Each iteration of the loop effectively adds A to X until $X = 0$ (A divides B) or $X < 0$ (A does not divide B). The resulting program P_2 may violate the constraint on the accumulator during this preprocessing phase, but only a constant number of times; however, the proof of Theorem 3.10 still applies. □

4. Nondeterministic reversal-bounded multicounter machines. We do not know if the emptiness problem for $\bigcup_r$ 2NCM(1, r) is decidable. In fact, the question is open even for $\bigcup_r$ 2NCM(1, r) machines accepting only bounded languages. However, we can show that the ECE problems are decidable for $\bigcup_c \bigcup_r$ 2NCM(c, r) on unary alphabet. More precisely, we prove that unary languages accepted by $\bigcup_c \bigcup_r$ 2NCM(c, r) machines are effectively regular. This settles a conjecture of Gurari and Ibarra [5]. In [5] it was only shown that unary languages accepted by $\bigcup_c \bigcup_r$ 2DCM(c, r) machines are effectively regular. Our technique is totally different. We first prove a lemma that characterizes regular unary languages.

LEMMA 4.1. *A unary language L is regular iff there is a constant c depending only on L such that for every $w = 0^x \in L$, $x > c$, there is some j, $1 \leq j \leq c$, such that $w' = 0^{jn}0^{x-j} \in L$ for all $n \geq 1$.*

Proof. The $\Rightarrow$ direction is immediate, since the stated condition is simply the condition of the pumping lemma for unary regular languages. Conversely, suppose L is a unary language

that satisfies the condition of the lemma. Partition L into c^2 subsets $L_{0,1}, L_{1,1}, \ldots, L_{c-1,c}$, so that $L_{i,j} = \{0^x \in L : j$ satisfies the condition for 0^x and $i = (x - j) \bmod j\}$. Then each $L_{i,j}$ is generated by its smallest element, and hence each $L_{i,j}$ and L is regular. □

THEOREM 4.2. *Every unary language accepted by a* 2NCM(c, r) *is regular for* $c, r \geq 1$.

Proof. It suffices to show the theorem for 2NCM(c, 1), since we can reduce the number of counter reversals by any 2NCM(c, r) to 1 by adding more counters. We first show the theorem for $c = 1$, using the characterization given in Lemma 4.1, and then extend the proof to the general case.

Suppose L is accepted by some 2NCM(1, 1) M with q states. To avoid stating unnecessary constants, we will use the word "bounded" in the following to mean "bounded by some constant depending only on q."

Let 0^x be in L, and let C be an accepting computation of M on 0^x. C has a *simple loop* if there are some $t_1 < t_2$ such that at times t_1 and t_2 M is in the same state and on the same input square, and furthermore M does not visit an endmarker during $[t_1, t_2]$. A simple loop is called *positive* or *negative* depending on the net change it makes to the counter value. C has a *sweep* if there are some $t_1 < t_2$ such that at times t_1 and t_2 M is on an endmarker, and M does not visit an endmarker during (t_1, t_2).

So given a computation C, we decompose C into sweeps of length $O(|0^x|)$ after first removing all simple loops. Since each boundary is crossed at most q times in each direction, the the number of different crossing sequences is bounded by $t = \sum_{i=0}^{2q}(2q)^i < (2q)^{2q+1}$, and so we can divide each sweep into a bounded number of clusters of identical movable segments (segments which have the same crossing sequences at both ends) separated by separator segments as explained in Theorem 2.1. Let b be the least common multiple of the lengths of all types of movable segments. After consolidating, we may assume that every movable segment has length b, and each sweep has exactly m movable segments for some $m = \Omega(|0^x|)$. (We consider sweeps that start and end on the same marker to have "empty" movable segments whose net change to the counter is zero.) We use $v(F)$ to denote the net change to the counter value by a computation fragment F (such as a loop or a sweep) of C. For example, $v(C) = 0$.

Suppose we add a segment of length b to the input 0^x. We can obtain from C an "almost" valid computation C' by duplicating one arbitrary movable segment in each sweep. (The simple loops are not affected since the input head never visits the endmarkers during a simple loop.) We call a set of movable segments obtained by taking one movable segment from each sweep a *cross section* E of C. A cross section E is called *positive* or *negative* depending on the sign of $v(E)$. Our strategy is to manipulate the sweeps, loops, and cross sections to obtain from C an accepting computation of M on 0^{x+kbn} for some bounded k and all $n \geq 1$.

We have the following seven cases.

1. C has a negative cross section and a positive cross section.

Let E^+ be the least positive cross section and E^- the least negative cross section. We claim $v(E^+)$ is bounded; otherwise, we can obtain another cross section E by replacing a movable segment in E by another choice in the same sweep with a smaller net change to the counter value. (We know such a sweep and a movable segment must exist, because there is a negative cross section.) We have $0 < v(E) < v(E^+)$, contradicting E^+'s minimality. The same applies for E^-. Letting $k = v(E^+) + |v(E^-)|$, we can obtain from C an accepting computation for 0^{x+kbn}, $n \geq 1$, by adding $|v(E^-)|n$ copies of cross section E^+ and $v(E^+)n$ copies of cross section E^-.

2. C has only positive cross sections, but also negative simple loops.

Let E be a positive cross section and K be a negative simple loop of C. From K we can obtain by cut-and-paste a new negative simple loop K' such that $v(K')$ is bounded. Letting

$k = |v(K')|$, we can obtain from C an accepting computation for 0^{x+kbn}, $n \geq 1$, by adding $|v(K')|n$ copies of cross section E and $v(E)n$ copies of simple loop K'.

3. C has only negative cross sections, but also positive simple loops. Symmetric to Case (2).

4. C has only positive cross sections, but also positive simple loops.

If C has only positive cross sections, then the number of sweeps in the decreasing phase cannot be bounded. To see this, recall that each sweep consists of m movable segments and some separator segments. Since C has only positive cross sections, we can choose any combination of movable segments (one in each sweep) and still have a positive cross section. Since there are $m \in \Omega(|0^x|)$ movable segments in each sweep, the net change to the counter values by the movable segments in all sweeps is $\Omega(|0^x|)$. If the number of sweeps in the decreasing phase is bounded, then the net change by the separator segments in those sweeps is $-O(1)$, not enough to make the final counter value of C zero. Hence C cannot be an accepting computation, a contradiction. So there is a sequence S of at most $2q + 1$ consecutive sweeps in the decreasing phase such that S starts and ends in the same state on the same endmarker, and $v(S) < 0$. Such a sequence is called a *sweep loop*.

Let E be a positive cross section and K be a positive simple loop of C. Again, we can assume that $v(K)$ is bounded. Let E' be the subset of movable segments of E that come from the sweeps in S, and let r be the least positive integer such that $v(K)v(E) + rv(K)(v(S) + v(K)v(E'))$ is negative. Letting $k = v(K)$, we can obtain from C an accepting computation for 0^{x+kbn}, $n \geq 1$, by adding $rv(K)n$ copies of the sweep loop S, $v(K)n$ copies of cross section E (E includes movable segments from the newly added copies of S), and $|v(E) + r(v(S) + v(K)v(E'))|n$ copies of simple loop K.

5. C has only negative cross sections, but also negative simple loops. Symmetric to case 4.

6. C has only positive cross sections, and no simple loops.

Since C has only positive cross section, again the number of sweeps in the decreasing phase cannot be bounded, and there is some negative sweep loop. We can assume that C has no sweep loops of net change $o(|0^x|)$. (Otherwise, we can obtain by cut-and-paste from such a sweep loop a new sweep loop S such that $v(S)$ is bounded. This sweep loop is in effect a simple loop, and we can proceed as in case 2 or 4.) Because the net change to the counter value in the decreasing phase is not bounded by $O(|0^x|)$ and there are no positive simple loops, the number of sweeps in the increasing phase cannot be bounded either, and there is also some positive sweep loop.

Let P be a positive sweep loop, and denote the net change by all separator segments in P by s_P. Select an arbitrary cross section of P (i.e., a set of movable segments in P, one from every sweep), and denote its net change by c_P. We can construct a new uniform sweep loop P' from P such that all cross sections of P' are the same, by replacing all movable segments in each sweep of P by the one selected for the cross section, and adding a extra copies, where a is a bounded number specified in the next paragraph. The net change by P' is $v(P') = (m + a)c_P + s_P$. Similarly, we can obtain a new uniform sweep loop N' from N such that the net change by N' is $v(N') = (m + a)c_N + s_N$, where c_N and s_N are defined analogously.

Now suppose we can find cross sections c_P and c_N such that $c_P s_N - c_N s_P$ is nonzero. Let a be $|c_P s_N - c_N s_P|$. Then $k = |gcd(v(P'), v(N'))|$ is bounded, since k must divide $c_P s_N - c_N s_P$, and c_P, c_N, s_P, s_N are all bounded. Hence, there are positive integers n_1, n_2 such that $n_1 v(P') + n_2 v(P') = -k$. In effect, we have a negative simple loop of net change $-k$. In this case, we can obtain from C an accepting computation for 0^{x+kbn}, $n \geq 1$, by adding $n_1 av(E)n/k$ copies of positive uniform sweep loop P', $n_2 av(E)n/k$ copies of negative

uniform sweep loop N', and *an* copies of a positive cross section E (E includes movable segments from the newly added sweep loops).

It remains to show that we can always choose the cross sections c_P and c_N so that $c_P s_N - c_N s_P$ is nonzero. Suppose not. Then all sweep loops in C must be uniform, i.e., there is only one unique cross section for each sweep loop, or else we can replace a single movable segment in one of the cross sections to make $c_P s_N - c_N s_P$ nonzero. Select arbitrarily a positive sweep loop J and a negative sweep loop K and discard $|c_K|$ copies of J and $|c_J|$ copies of K from C. Do this until it is no longer possible. At this point, either the number of positive sweep loops or the number of negative sweep loops is bounded. Note that the net change by all discarded sweep loops is zero, by our assumption.

If the number of negative sweep loops is bounded, then in fact the net change by all separator segments in C is $-O(1)$, not enough to offset the net change by all movable segments in C, which is $\Omega(|0^x|)$. So C cannot be an accepting computation, a contradiction. If the number of negative sweep loops is not bounded, then since the number of positive sweep loops is bounded, the total net change by the remaining sweeps must be negative (we noted earlier that the net change by a negative sweep loop is $-|\Omega(|0^x|)|$). Again, C cannot be an accepting computation, a contradiction.

7. C has only negative cross sections, and no simple loops. Symmetric to Case (6).

In all cases, we can find some bounded k such that 0^{x+kbn} is in L for all $n \geq 1$. Hence by Lemma 4.1, L is regular. This concludes the proof of the theorem for $c = 1$.

Now we show how to extend the above proof to the general case. Suppose L is accepted by some 2NCM(c, 1) M. Let 0^x be in L, and let C be an accepting computation of M on 0^x. We partition C into simple loops, movable segments, and sweeps as for the case of 2NCM(1, 1), except that now these units are defined relative to a single counter. So there are c different partitions of C, each concerning with net changes to only one counter and ignoring the other counters. Proceed as in the construction above to find the size of the segment to be added for each case, say $k_1 b$, $k_2 b$, ..., $k_c b$. Then the final input segment to be added is $k_1 k_2 \ldots k_c b^c$. □

The proof of Theorem 4.2 also works even if we shorten instead of lengthen the input. (In this case, we need to remove cross sections and simple loops but still *add* sweeps. We adjust the definitions to make sure that cross sections and simple loops always exist in multiple copies.) Thus, to decide whether a 2NCM(c, r) accept some input, we only need to check all inputs of length at most a bounded constant. Since the membership problem for 2NCM(c, r) is decidable [2], we have the following.

COROLLARY 4.3. *The ECE problems for* $\bigcup_c \bigcup_r$ 2NCM(c, r) *on unary alphabet are decidable.*

This is the strongest result one can obtain since it is known that deterministic reversal-bounded 2-counter machines over strictly bounded languages can accept nonsemilinear sets and have an undecidable emptiness problem [8].

REFERENCES

[1] R. ALUR, T. HENZINGER, AND M. VARDI, *Parametric real-time reasoning*, in Proc. 25th Annual ACM Symposium on the Theory of Computing, San Diego, California, ACM Press, 1993, pp. 592–601.

[2] T.-H. CHAN, *Reversal complexity of counter machines*, in Proc. 13th Symp. on Theory of Computing, ACM, Milwaukee, Wisconsin, ACM Press, 1981, pp. 146–157.

[3] ———, *On two-way weak counter machines*, Math. System Theory, 20 (1987), pp. 31–41.

[4] M. DAVIS, Y. MATIJASEVIČ, AND J. ROBINSON, *Hilbert's tenth problem. Diophantine equations: Positive aspects of a negative solution*, Proc. Sympos. Pure Math., 28 (1976), pp. 323–378.

[5] E. M. GURARI AND O. H. IBARRA, *Simple counter machines and number-theoretic problems*, J. Comput. System Sci., 19 (1979), pp. 145–162.

[6] E. M. GURARI AND O. H. IBARRA, *Two-way counter machines and diophantine equations*, J. Assoc. Comput. Mach., 29 (1982), pp. 863–873.

[7] O. IBARRA, T. JIANG, N. TRAN, AND H. WANG, *On the equivalence of two-way pushdown automata and counter machines over bounded languages*, in Proc. of the 10th Symposium on Theoretical Aspects of Computer Science, Würzburg, Germany, Springer-Verlag, Berlin, 1993, pp. 354–364.

[8] O. H. IBARRA, *Reversal-bounded multicounter machines and their decision problems*, J. Assoc. Comput. Mach., 25 (1978), pp. 116–133.

[9] M. MINSKY, *Recursive unsolvability of Post's problem of tag and other topics in the theory of Turing machines*, Ann. of Math., 74 (1961), pp. 437–455.

[10] M. SIPSER, *Halting space-bounded computations*, Theoret. Comput. Sci., 10 (1980), pp. 335–338.

[11] L. G. VALIANT AND M. S. PATERSON, *Deterministic one-counter automata*, J. Comput. System Sci., 10 (1975), pp. 340–350.

SIAM J. COMPUT.
Vol. 24, No. 1, pp. 138–147, February 1995

009

ON THE OPTIMALITY OF RANDOMIZED α-β SEARCH *

YANJUN ZHANG†

Abstract. It is shown that the expected number of leaves evaluated by randomized α-β search for evaluating uniform game trees of degree d and height h is $O((B_d)^h)$, where $B_d = d/2 + \ln d + O(1)$. It was shown by Saks and Wigderson [*Proceedings of* 27th *Annual Symposium on Foundations of Computer Science* (1986), pp. 29–38] that the optimal branching factor of randomized algorithms for evaluating uniform trees of degree d is $B_d^* = (d - 1 + \sqrt{d^2 + 14d + 1})/4 = d/2 + O(1)$. As $B_d/B_d^* = 1 + O(\ln d/d)$, randomized α-β search is asymptotically optimal for evaluating uniform game trees as the degree of tree increases.

Key words. game-tree evaluation, α-β search, randomized algorithms

AMS subject classifications. 68Q25, 68T20

1. Introduction.

A *game tree* is a rooted tree in which the root is a MAX-node, the internal nodes at odd (even) distance from the root are MIN-nodes (MAX-nodes), and each leaf has a real value. The *value*, or *minimax value*, of a MAX-node (MIN-node) is recursively defined as the maximum (minimum) of the values of its children. The problem of game-tree evaluation is that of determining the value of the root of a game tree. Game-tree evaluation is used in computer programs for playing two-person games of perfect information such as chess and checkers.

The best known heuristic in practice for evaluating game trees is the α-β pruning procedure [4], which we shall call α-β *search*. The efficiency of α-β search lies in its ability to detect and prune away certain nodes whose values can no longer influence the value of the root. A *uniform* tree is a tree in which each internal node has the same number of children and each root-leaf path has the same number of nodes; for a uniform tree, the *degree* is the number of children of an internal node, and the *height* is the number of edges on a root-leaf path. Traditionally, the performance of α-β search has been analyzed on uniform game trees in the i.i.d. probabilistic model in which the values of leaves are drawn independently from the distribution of a certain random variable, and the complexity of a game-tree evaluation algorithm is the expected number of leaves evaluated by the algorithm. The *branching factor* of an algorithm is the asymptotic limit of the hth root of the expected number of leaves evaluated by the algorithm as the height of tree h increases.

It was shown by Knuth and Moore [4] that in the i.i.d. model the branching factor of a weaker version of α-β search using only "shallow cutoff" is $\Theta(d/\ln d)$ on uniform trees of degree d, and conjectured that the branching factor of α-β search remains $\Theta(d/\ln d)$. Their conjecture was confirmed by Baudet [1] (cf. [7, p. 294]). The exact value of the branching factor of α-β search on random uniform trees of degree d was shown by Pearl [6] to be $R_d^* = \frac{\xi_d}{1-\xi_d}$, where ξ_d is the positive root of $x^d + x - 1 = 0$ and $R_d^* = \frac{d}{\ln d}(1 + O(\frac{\ln\ln d}{\ln d}))$ (cf. [7, p. 267]). A year later, Tarsi [9] proved that branching factor R_d^* is best possible for evaluating random uniform trees in the i.i.d. model.

More recently, Saks and Wigderson [8] studied the game-tree evaluation by randomized algorithms. A randomized game-tree evaluation algorithm can make random choices during its execution in deciding which leaf to evaluate next, and the number of leaves evaluated on a fixed input instance is a random variable. The complexity of a randomized game-tree evaluation algorithm is the maximum of the expected number of leaves evaluated over all instances of game trees. The randomized complexity for game-tree evaluation is the minimum

*Received by the editors May 4, 1992; accepted for publication (in revised form) September 23, 1993.

†Department of Computer Science and Engineering, Southern Methodist University, Dallas, Texas 75275 (zhang@seas.smu.edu). The research of this author was supported in part by NSF grant CCR-9110839.

of the complexity over all randomized game-tree evaluation algorithms. The chief advantage of randomized algorithms is that no probabilistic assumptions are imposed upon the input instances, as probabilistic assumptions may fail to reflect reality. Saks and Wigderson proved in [8] that the randomized complexity for evaluating uniform game trees of degree d and height h is $\Theta((B_d^*)^h)$, where $B_d^* = \left(d - 1 + \sqrt{d^2 + 14d + 1}\right)/4 = d/2 + 3/2 + O(d^{-1})$ is the optimal branching factor in the randomized model. The quantitative difference in optimal branching factors $B_d^* = d/2 + O(1)$ of the randomized model and $R_d^* = \Theta(d/\ln d)$ of the i.i.d. model demonstrates a fundamental difference in the two probabilistic models. Furthermore, they show that the optimal upper bound of $O((B_d^*)^h)$ is achieved by a randomized version of SCOUT, a game-tree evaluation algorithm studied by Pearl [5].[1] SCOUT is a quite different algorithm from α-β search; there is no dominating relation between SCOUT and α-β search in terms of the number of evaluated leaves. (See [7, pp. 246–250] for such examples.) The question of whether randomized α-β search is also optimal was not resolved by the work of Saks and Wigderson.

In this paper we show that randomized α-β search is asymptotically optimal for evaluating uniform game trees as the degree of tree increases. We show that the expected number of leaves evaluated by randomized α-β search for evaluating uniform game trees of degree d and height h is $O((B_d)^h)$, where $B_d = d/2 + O(\ln d)$, and $B_d/B_d^* = 1 + O(\ln d/d)$, where $B_d^* = d/2 + O(1)$ is the optimal branching factor. Our approach is similar to that used by Knuth and Moore for the analysis of α-β search in the i.i.d. model. Instead of a direct analysis of randomized α-β search, we shall analyze a weaker version of randomized α-β search using only shallow cutoff. We show that the randomized α-β search using only shallow cutoff achieves the claimed bound, thus asymptotically optimal as the degree of tree increases. It follows that randomized α-β search is also asymptotically optimal.

2. α-β search. The execution of α-β *search*[2] is a depth-first traversal of the input tree during which certain nodes are evaluated and others are pruned away, thus not evaluated. The pruned nodes are certified to have no influence on the value of the root. Let $\mathrm{val}(v)$ be the minimax value of node v. Each visited node v is evaluated upon the return of the traversal from the subtree rooted at that node with a *return value* $r(v)$, which may or may not be equal to $\mathrm{val}(v)$. The return value $r(v)$ is recursively defined as the leaf value of v if v is a leaf or the maximum (minimum) of the return values of the evaluated children of v if v is a MAX-node (MIN-node). The depth-first traversal of α-β search visits the children of a node in the left-to-right order. The *randomized* α-β *search* is an α-β search in which the children of a node are visited in a random order, independent of the order in which other nodes are visited.

The pruning mechanism of α-β search consists of two parameters, the α bound and the β bound. The *current value* of a MAX-node (MIN-node) v, denoted by $c(v)$, is the maximum (minimum) of the return values of the evaluated children of v. The current value of a node becomes its return value upon the evaluation of that node. Let $c(v) = -\infty(+\infty)$ for a MAX-node (MIN-node) v with no evaluated children. An *ancestor* of v is a node on the path between v and

[1] For the sake of completeness, we describe SCOUT in this footnote. SCOUT is a two-pass evaluation process that tests the usefulness of a node before deciding to evaluate or discard it. To evaluate a node v, SCOUT recursively evaluates a child u of v and sets $c(v) = \mathrm{val}(u)$. For each remaining child u' of v, SCOUT first tests whether $\mathrm{val}(u') > c(v)$ or $\mathrm{val}(u') < c(v)$ without determining the exact value of $\mathrm{val}(u')$; such a test can be done as an evaluation of an AND/OR tree of Boolean values. If $\mathrm{val}(u') > c(v)$ for a MAX-node v or $\mathrm{val}(u') < c(v)$ for a MIN-node v, then recursively evaluate u' and set $c(v) = \mathrm{val}(u')$; otherwise, discard u' without further evaluation. After all children of v are processed, $c(v)$ is returned (correctly) as $\mathrm{val}(v)$. SCOUT is also optimal in the i.i.d. model [7]. The randomized SCOUT in [8] is to randomly select the next child for testing and evaluation.

[2] In [4], α-β search is described by a recursive program $F2$ in the negmax format; the correctness of α-β search is proved via an invariant argument about program $F2$. For a node at which pruning occurs, $F2$ may compute a different value than the return value of that node defined in this paper.

the root, including v itself. The α *bound* of v is $\alpha(v) = \max\{c(w) \mid w \text{ is a MAX-ancestor of } v\}$ and the β *bound* of v is $\beta(v) = \min\{c(w) \mid w \text{ is a MIN-ancestor of } v\}$. During its traversal, α-β search maintains the α bound and β bound of the currently visited node. The *pruning rule* of α-β search is that the unevaluated children of a node v are pruned if $\alpha(v) \geq \beta(v)$. Pruning *occurs at* v if the unevaluated children of v are pruned but v itself is not. All children of the root r are evaluated, as r is a MAX-node and $\beta(r) = +\infty$ always.

In [4], Knuth and Moore described a weaker version of α-β search using only "shallow cutoff," which is given by program $F1$ together with a proof of correctness, and analyzed its asymptotic behavior for evaluating uniform random trees in the i.i.d. model. For any node v, define the *shallow* α*-bound* $\alpha'(v)$ and the *shallow* β*-bound* $\beta'(v)$ of v as follows. For root r, let $\alpha'(r) = c(r)$ and $\beta'(r) = +\infty$. For a MAX-node or MIN-node $v \neq r$, let w be the parent of v, and define $\alpha'(v) = c(v)$ and $\beta'(v) = c(w)$ if v is a MAX-node, or $\alpha'(v) = c(w)$ and $\beta'(v) = c(v)$ if v is a MIN-node. By definition, $\alpha'(v) \leq \alpha(v)$ and $\beta'(v) \geq \beta(v)$. Since $\alpha'(v) \leq \alpha(v)$ and $\beta'(v) \geq \beta(v)$, condition $\alpha'(v) \geq \beta'(v)$ guarantees the pruning criterion $\alpha(v) \geq \beta(v)$. The pruning occurring at v is called a *shallow cutoff* if $\alpha'(v) \geq \beta'(v)$; otherwise, it is a *deep cutoff*. Hence, the pruning at v is a shallow cutoff when pruning criterion $\alpha(v) \geq \beta(v)$ is satisfied by the current values of v and the parent of v. Figure 1 illustrates a shallow cutoff and a deep cutoff for game trees of degree 2. A basic fact about shallow cutoff is that a shallow cutoff cannot occur at a node that is evaluated first among its siblings. This is because the current value of the parent node of v will be either ∞ or $-\infty$ when v is evaluated first among its siblings, rendering the shallow cutoff condition $\alpha'(v) \geq \beta'(v)$ impossible. This fact will be used repeatedly in the later analysis.

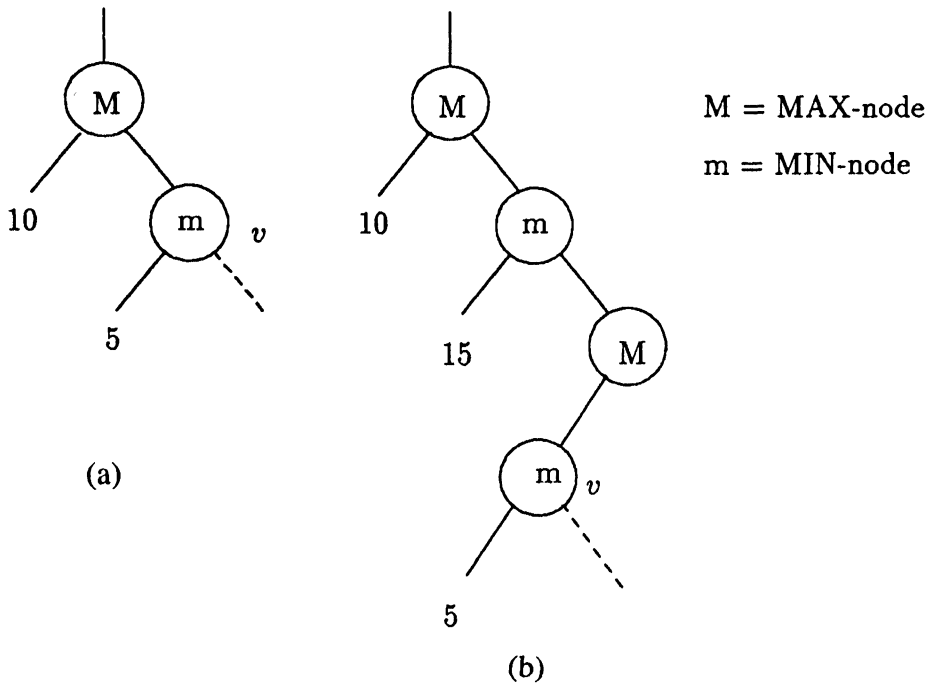


FIG. 1. (a) *a shallow cut at* v; (b) *a deep cut at* v. *In both* (a) *and* (b), $\alpha(v) \geq 10 > 5 \geq \beta(v)$.

We call the α-β search performing only shallow cutoffs *shallow* α-β *search*. A shallow α-β search in which the children of a node are visited in a random order is called *randomized*

shallow α-β search. We shall study shallow α-β search and randomized shallow α-β search in the next two sections.

We end this section with a useful characterization of α-β search. For a leaf u of a game tree, let $A(u)$ be the maximum of the values of all left siblings of a MIN-ancestor of u, and let $B(u)$ be the minimum of the values of all left siblings of a MAX-ancestor of u. Notice that $A(u)$ and $B(u)$ depend *statically* on the structure of the game tree and the minimax values of nodes. The following criterion for a leaf u to be evaluated by α-β search in terms of $A(u)$ and $B(u)$, in a similar way as the α bound and the β bound, was first shown in [2] and later in [1] (cf. [7, p. 239]).

THEOREM 1. *A leaf u is evaluated by α-β search if and only if $A(u) < B(u)$.*

3. Shallow α-β search. In this section we prove some basic properties of shallow α-β search, and show the domination of (randomized) α-β search over (randomized) shallow α-β search.

A fundamental relation between the return value $r(v)$, the minimax value $\mathrm{val}(v)$, and shallow bounds $\alpha'(v)$ and $\beta'(v)$ for any node v evaluated by shallow α-β search is given in Proposition 1. We note that neither (i) and (ii) of Proposition 1 holds in general for α-β search when deep cutoffs are used. The proof of Proposition 1 is not as straightforward as one may think, due to the fact that the return value of a node may differ from its minimax value. Proposition 1 also provides a proof of correctness of shallow α-β search.

PROPOSITION 1. *Suppose that a nonleaf node v is evaluated in a shallow α-β search. Then* (i) *if no pruning occurs at v, then* $\mathrm{val}(v) = r(v)$; *in particular,* $\mathrm{val}(r) = r(r)$ *where r is the root*; (ii) *if pruning occurs at v, then* $\mathrm{val}(v) \geq r(v) \geq \beta'(v)$ *if v is a MAX-node or* $\mathrm{val}(v) \leq r(v) \leq \alpha'(v)$ *if v is a MIN-node, where $\alpha'(v)$ and $\beta'(v)$ are, respectively, the shallow α bound and shallow β bound of v before the evaluation of v.*

Proof. We assume that v is a MAX-node. The case of a MIN-node v is symmetric. If pruning occurs at v, by the pruning rule, $r(v) = \alpha'(v) \geq \beta'(v)$, where $\beta'(v)$ is the current value of the parent of v. Observe that $\beta'(v)$ does not change during the evaluation of v, thus can be taken as being computed before the evaluation of v. This proves the second inequality $r(v) \geq \beta'(v)$ of (ii).

We shall prove (i) and the first inequality of (ii) by induction on the order of evaluation. Notice that (i) and (ii) hold for any evaluated leaf. Suppose that we have just evaluated v. We want to show that (i) and (ii) hold for v. Let $u_1, u_2, \ldots, u_{d'}$ be the set of evaluated children of v that were evaluated in that order. By induction, (i) and (ii) hold for $u_1, u_2, \ldots, u_{d'}$. Define a sequence of indices $i_j \leq d'$ as follows. Let $i_1 = 1$, and for $j \geq 1$, $i_{j+1} = \min\{k \leq d' \mid \mathrm{val}(u_k) > \mathrm{val}(u_{i_j})\}$, and i^* be the last i_j. By definition, $\mathrm{val}(u_{i_1}) < \mathrm{val}(u_{i_2}) < \cdots < \mathrm{val}(u_{i^*}) \leq \mathrm{val}(v)$, and if no pruning occurs at v, $\mathrm{val}(u_{i^*}) = \mathrm{val}(v)$. Hence, (i) and the first inequality of (ii) follow immediately if

$$r(v) = \mathrm{val}(u_{i^*}). \tag{1}$$

We shall prove the following statements (a) and (b) from which (1) follows immediately, given $r(v) = \max_{1 \leq k \leq d'}\{r(u_k)\}$.

(a) For $j \geq 1$, $\mathrm{val}(u_{i_j}) = r(u_{i_j})$.

(b) For $i_j < k < i_{j+1}$, $r(u_k) \leq r(u_{i_j})$ and for $i^* < k$, $r(u_k) \leq r(u_{i^*})$.

We prove (a) and (b) by induction on $k = 1, 2 \ldots, d'$. The basis of the induction is to show that $\mathrm{val}(u_1) = r(u_1)$. As u_1 is the first evaluated child of v, no pruning may occur at u_1. By induction of (i) on u_1, $\mathrm{val}(u_1) = r(u_1)$. We now assume inductively that (a) and (b) hold for all $u_{k'}$, where $k' < k$. Consider the MIN-node u_k. The shallow bound $\alpha'(u_k)$ before the evaluation of u_k is $\alpha'(u_k) = c(v) = \max_{k' < k}\{r(u_{k'})\}$. By induction of (a) and (b)

on $u_1, u_2, \ldots, u_{k-1}$, and the definition of indices i_j,

$$\alpha'(u_k) = \max_{k'<k}\{r(u_{k'})\} = \begin{cases} r(u_{i_j}) = \text{val}(u_{i_j}) & \text{if } i_j < k \le i_{j+1}, \\ r(u_{i^*}) = \text{val}(u_{i^*}) & \text{if } i^* < k. \end{cases} \tag{2}$$

Consider three cases of k.

Case 1. $k = i_{j+1}$, which is (a). In this case, we need to show $\text{val}(u_k) = r(u_k)$. By (2), $\alpha'(u_k) = \text{val}(u_{i_j}) < \text{val}(u_{i_{j+1}}) = \text{val}(u_k)$ before the evaluation of u_k. If pruning occurs at u_k, by induction of (ii) on u_k, we would have $\alpha'(u_k) \ge \text{val}(u_k)$—a contradiction. Hence, no pruning occurs at u_k, and by induction of (i) on u_k, $\text{val}(u_k) = r(u_k)$.

Case 2. $i_j < k < i_{j+1}$, the first case of (b). In this case, we need to show $r(u_k) \le r(u_{i_j})$. If no pruning occurs at u_k, by induction of (i), $\text{val}(u_k) = r(u_k)$. Since $i_j < k < i_{j+1}$, $\text{val}(u_k) \le \text{val}(u_{i_j}) = r(u_{i_j})$ by induction of (a) on u_{i_j}. So $r(u_k) \le r(u_{i_j})$ as desired. If pruning occurs at u_k, by induction of (ii), $r(u_k) \le \alpha'(u_k)$ before evaluation of u_k. By (2), $\alpha'(u_k) = r(u_{i_j})$. So again, $r(u_k) \le r(u_{i_j})$.

Case 3. $i^* < k$, the second case of (b). This case is identical to Case 2 with i^* in place of i_j. The induction on (a) and (b) is complete. □

The following proposition states that in shallow α-β search the current value of a node can be expressed in terms of minimax values of its evaluated children instead of their return values. This fact is critical to the later analysis.

PROPOSITION 2. *Let $D(v)$ be the set of children of v evaluated by shallow α-β search thus far. Suppose that $|D(v)| > 0$. Then $c(v) = \max_{u\in D(v)}\{\text{val}(u)\}$ if v is a MAX-node, or $c(v) = \min_{u\in D(v)}\{\text{val}(u)\}$ if v is a MIN-node, where $c(v)$ is the current value of v. Consequently, $c(v) \le \text{val}(v)$ if v is a MAX-node, or $c(v) \ge \text{val}(v)$ if v is a MIN-node.*

Proof. Assume that v is a MAX-node. We use induction on $|D(v)|$. In the basis case $|D(v)| = 1$, no pruning occurs at the first evaluated child u of v. By Proposition 1(i), $c(v) = r(u) = \text{val}(u)$. Assume inductively that $c(v) = \max_{u\in D(v)}\{\text{val}(u)\}$. Let MIN-node u' be the next evaluated child of v. By Proposition 1, if no pruning occurs at u', then $r(u') = \text{val}(u')$; otherwise, $\alpha'(u') = c(v) \ge r(u') \ge \text{val}(u')$. In either case, let $c'(v)$ be the current value of v after u' is evaluated, and we have $c'(v) = \max\{c(v), r(u')\} = \max\{c(v), \text{val}(u')\} = \max_{u\in D'(v)}\{\text{val}(u)\}$ where $D'(v) = D(v) \cup \{u'\}$. □

One can derive a criterion for a leaf u to be evaluated by shallow α-β search in terms of the static structure of the tree, similar to the one given in Theorem 1 in §2. Let v be an ancestor of leaf u. Define $A(v, u)$ (respectively, $B(v, u)$) to be the maximum (minimum) of the minimax values of the children of v that are left siblings to the child of v that is the ancestor of u if v is a MAX-node (MIN-node).

PROPOSITION 3. *A leaf u is not evaluated by shallow α-β search if and only if there are a MAX ancestor v of u and a MIN ancestor w of u where w is the parent of v or v is the parent of w such that $A(v, u) \ge B(w, u)$.*

Proof. Suppose that u is not evaluated by shallow α-β search. Let v be the ancestor of u at which pruning occurs. We may assume that v is a MAX-node. Upon pruning at v, $c(v) = \alpha'(v) \ge \beta'(v) = c(w)$, where w is the MIN parent of v, and the evaluated children of w (respectively, v) are exactly the left siblings of v (respectively, v', where v' is the child of v that is an ancestor of u). By Proposition 2, $c(v) = A(v, u)$ and $c(w) = B(w, u)$. Hence, $A(v, u) \ge B(w, u)$. On the other hand, suppose that u is evaluated by shallow α-β search. Let v be a MAX-ancestor of u and w a MIN-ancestor of u such that w is the parent of v or vice versa. Upon the evaluation of u, by Proposition 2, $A(v, u) = c(v) = \alpha'(v) < \beta'(v) = c(w) = B(w, u)$ if w is the parent of v, or $A(v, u) = c(v) = \alpha'(w) < \beta'(w) = c(w) = B(w, u)$ if v is the parent of w. In ether case, we have $A(v, u) < B(w, u)$. □

As a corollary, we conclude that α-β search performs no worse than shallow α-β search on any instance. This fact, though taken for granted in the literature, is not obvious because

the return values computed by shallow α-β search may be different than those computed by α-β search.

COROLLARY 1. *For any game tree T, we evaluate T by α-β search and again by shallow α-β search. If a leaf u is evaluated by α-β search, then u is also evaluated by shallow α-β search.*

Proof. The proof follows immediately from Proposition 3, Theorem 1, and the fact that $A(v, u) \leq A(u)$ and $B(v, u) \geq B(u)$. □

COROLLARY 2. *For any game tree T, let $R(T)$ and $R'(T)$ be the expected number of leaves evaluated by randomized α-β search and randomized shallow α-β search, respectively. Then $R(T) \leq R'(T)$.*

Proof. The proof follows immediately from Corollary 1 and the fact that the execution of randomized (shallow) α-β search is equivalent to the execution of deterministic (shallow) α-β search on the randomly permuted input tree. □

4. Randomized shallow α-β search. In this section we analyze the complexity of randomized shallow α-β search for evaluating uniform game trees. We show that the expected number of leaves evaluated by randomized shallow α-β search on any instance of uniform game tree of degree d and height h is $\Theta((B_d)^h)$, where $B_d = d/2 + \ln d + O(1)$.

We first state two probability facts about random permutations. In a permutation $(a_1, a_2, \ldots, a_n)$ of $\{1, 2, \ldots, n\}$, a_i is a *left-to-right maximum* if $a_i > a_j$ for all $j < i$. In other words, a_i is a left-to-right maximum if and only if position i contains the maximum among the values assigned to position $1, 2, \ldots, i$. For example, in permutation $(a_1, a_2, a_3, a_4, a_5) = (2, 1, 4, 5, 3)$, the left-to-right maxima are $a_1 = 2$, $a_3 = 4$ and $a_4 = 5$.

LEMMA 1. *The expected number of left-to-right maxima in a random permutation of $\{1, 2, \ldots, n\}$ is the nth harmonic number $H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$.*

Proof. The probability that position i of a random permutation contains the maximum among the values assigned to positions $1, 2, \ldots, i$ is $1/i$, as the maximum of these values is equally likely to be at any position of $1, 2, \ldots, i$. Hence, the expected number of left-to-right maxima is $H_n = \sum_{k=1}^{n} 1/k$. □

LEMMA 2. *In a random permutation of $n - 1$ white balls and one black ball, let W be the number of white balls proceeding the black ball in the permutation. Then $\Pr[W = k] = 1/n$ for $0 \leq k \leq n - 1$ and $E[W] = (n - 1)/2$.*

Proof. The black ball is equally likely to be at any position in a random permutation. The lemma follows. □

A critical notion in our analysis is that of "pruning information." By Proposition 2, $\alpha'(v) = c(v) \leq \text{val}(v)$ for a MAX-node v and $\beta'(v) = c(v) \geq \text{val}(v)$ for a MIN-node v, where $\alpha'(v)$ and $\beta'(v)$ are computed before evaluation of v. To meet the pruning condition $\alpha'(v) \geq \beta'(v)$, it must be the case that $\beta'(v) \leq \text{val}(v)$ for a MAX-node v and $\alpha'(v) \geq \text{val}(v)$ for a MIN-node v. We say v is *evaluated with pruning information* if, at the beginning of evaluation of v, $\beta'(v) \leq \text{val}(v)$ for a MAX-node v and $\alpha'(v) \geq \text{val}(v)$ for a MIN-node v; otherwise, v is *evaluated with no pruning information.* By Proposition 2, pruning may occur at v only if v is evaluated with pruning information. The root is evaluated with no pruning information. Clearly, pruning information may only help reduce the number of leaves evaluated. The following proposition characterizes the notion of pruning information.

PROPOSITION 4. *A nonleaf node v is evaluated with pruning information if and only if some sibling u of v has been evaluated such that $\text{val}(u) \leq \text{val}(v)$ if v is a MAX-node, or $\text{val}(u) \geq \text{val}(v)$ if v is a MIN-node.*

Proof. Suppose that a nonleaf v is evaluated with pruning information. Let w be the parent of v. Assume that v is a MAX-node. Then $c(w) = \beta'(v) \leq \text{val}(v)$. By Proposition 2,

$c(w) = \min\{\text{val}(u) | u \text{ is an evaluated sibling of } v\}$. Some sibling u of v with $\text{val}(u) \leq \text{val}(v)$ must have been evaluated. □

Let f_n (respectively, g_n) denote the maximum of expected number of leaves evaluated by randomized shallow α-β search on subtree H of height n over all subtrees H of height n in $T(d, h)$ under the condition that the the root of H is evaluated with no pruning information (respectively, with pruning information). By definition, $R'(T(d, h)) = f_h$ and $f_n \geq g_n$ for $0 \leq n \leq h$ with $f_0 = g_0 = 1$. We prove the following recurrence relation.

PROPOSITION 5. *For any n,* $1 \leq n \leq h$,

$$f_n \leq H_d f_{n-1} + (d - H_d) g_{n-1}, \tag{3}$$

$$g_n \leq (H'_d + 1) f_{n-1} + \left(\frac{d-1}{2} - H'_d\right) g_{n-1}, \tag{4}$$

where $H_d = 1 + \frac{1}{2} + \cdots \frac{1}{d}$ *and* $H'_d = \frac{1}{d}(H_1 + H_2 + \cdots + H_{d-1})$.

Proof. Let v be the root of a subtree of height n in $T(d, h)$. We may assume that v is a MAX-node. Let $u_1, u_2, \ldots, u_d$ be all d children of v arranged in the order such that $\text{val}(u_1) \leq \text{val}(u_2) \leq \cdots \leq \text{val}(u_d) = \text{val}(v)$. Suppose that v is evaluated with no pruning information. Then all children of v are evaluated. By Proposition 4, the child u_i, which is a MIN-node, is evaluated with no pruning information only if no sibling u of u_i with $\text{val}(u) \geq \text{val}(u_i)$ has already been evaluated, i.e., none of $u_{i+1}, u_{i+1}, \ldots, u_d$ and no u_j such that $j < i$ and $\text{val}(u_j) = \text{val}(u_i)$ has been evaluated. Hence, if u_i is evaluated with no pruning information, then u_i corresponds to a left-to-right maximum in a random permutation of $\{1, 2, \ldots, d\}$. Let R be the expected number of children of v evaluated with no pruning information. By Lemma 1, $R \leq H_d$. The expected number of children of v evaluated with pruning information is thus $d - R$. We have the following recurrence relation, noting $R \leq H_d$, $f_n \geq g_n$ and $(H_d - R)g_n \leq (H_d - R)f_n$:

$$\begin{aligned} f_n &\leq R f_{n-1} + (d - R) g_{n-1} \\ &= H_d f_{n-1} + (d - H_d) g_{n-1} - (H_d - R) f_{n-1} + (H_d - R) g_{n-1} \\ &\leq H_d f_{n-1} + (d - H_d) g_{n-1}, \end{aligned}$$

which is (3).

To analyze g_n, suppose that the MAX-node v is evaluated with pruning information. Then $\beta'(v) \leq \text{val}(v)$. Let $B = \{u_i \mid \text{val}(u_i) \geq \beta'(v)\}$, where u_i is a child of v, which is a nonempty set as $u_d \in B$ with $\text{val}(u_d) = \text{val}(v) \geq \beta'(v)$. By Proposition 2, $\alpha'(v) = c(v) = \max\{\text{val}(u) \mid u \text{ is an evaluated child of } v\}$. The pruning condition $\alpha'(v) \geq \beta'(v)$ holds if and only if some child of v in B is evaluated. Exactly one node in B is evaluated as the last evaluated child of v. Let W be the number of evaluated children of v before a child of v in B is evaluated. Viewing children of v in B as black balls and the other children of v as white balls in Lemma 2, we have $\Pr[W = k] \leq 1/d$ for $0 \leq k \leq d - 1$ and $E[W] \leq (d - 1)/2$.

The only evaluated child u in B, which is a MIN-node, is evaluated with no pruning information as, prior to the evaluation of u, $\alpha'(u) = c(v) = \max\{\text{val}(u_i) \mid u_i \text{ is an evaluated child of } v \text{ not in } B\} < \beta'(v) \leq \text{val}(u)$. Among the W evaluated children of v not in B, let R' be the expected number of nodes evaluated with no pruning information. Like we have argued previously, each of these evaluated nodes corresponds to a left-to-right maximum in a random permutation of $\{1, 2, \ldots, W\}$. By Lemma 1, $R' \leq H_k$ if $W = k$, and by conditioning on W,

$$R' \leq \sum_{k=1}^{d-1} H_k \Pr[W = k] \leq \frac{1}{d} \sum_{k=1}^{d-1} H_k = H'_d.$$

The expected number of children of v evaluated with pruning information is thus $E[W] - R'$. We have the following recurrence relation, noting $E[W] \leq (d-1)/2$, $R' \leq H'_d$, $f_n \geq g_n$,

$$\begin{aligned} g_n &\leq (R'+1)f_{n-1} + (E[W] - R')g_{n-1} \\ &= (H'_d + 1)f_{n-1} + (E[W] - H'_d)g_{n-1} - (H'_d - R')f_{n-1} + (H'_d - R')g_{n-1} \\ &\leq (H'_d + 1)f_{n-1} + \left(\frac{d-1}{2} - H'_d\right) g_{n-1}, \end{aligned}$$

which is (4). □

THEOREM 2. *Let $T(d,h)$ be any instance of a uniform game tree of degree $d \geq 2$ and height $h \geq 2$. Let $R'(T(d,h))$ be the expected number of leaves evaluated by randomized shallow α-β search for evaluating $T(d,h)$. Then*

$$R'(T(d,h)) = O((B_d)^h),$$

where, as a function of d,

$$B_d = \frac{d}{2} + \ln d + O(1).$$

Proof. Define $\lambda = H_d = \sum_{k=1}^{d} \frac{1}{k}$, $\mu = d - \lambda$, $\nu = 1 + H'_d = 1 + \frac{1}{d}\sum_{k=1}^{d-1} H_k$, and $\xi = \frac{d+1}{2} - \nu$. We rewrite the recurrence relations (3) and (4) in Proposition 5 as

$$\begin{aligned} f_n &\leq \lambda f_{n-1} + \mu g_{n-1}, \\ g_n &\leq \nu f_{n-1} + \xi g_{n-1}. \end{aligned}$$

Define $F_0 = 1 = f_0$, $G_0 = 1 = g_0$, and for $1 \leq n \leq h$,

$$F_n = \lambda F_{n-1} + \mu G_{n-1}, \tag{5}$$

$$G_n = \nu F_{n-1} + \xi G_{n-1}. \tag{6}$$

By a straightforward induction on n, we have for $0 \leq n \leq h$,

$$f_n \leq F_n. \tag{7}$$

By substituting (6) into (5), and noting $\mu G_{n-2} = F_{n-1} - \lambda F_{n-2}$ by (5), we obtain that for $1 \leq n \leq h$,

$$F_n = (\lambda + \xi)F_{n-1} + (\mu\nu - \lambda\xi)F_{n-2}. \tag{8}$$

The solution to recurrence relation (8) (cf. [3, §7.3]) is

$$F_n = \Theta((B_d)^n), \tag{9}$$

where B_d is the maximum of the two roots of the characteristic equation $x^2 = (\lambda + \xi)x + (\mu\nu - \lambda\xi)$ of recurrence relation (8).

To compute B_d, we calculate the following asymptotics. The harmonic number $H_d = \sum_{k=1}^{d} \frac{1}{k} = \ln d + \gamma + O(d^{-1})$, where $\gamma = .5772\ldots$, is the Euler's constant. Since $d! \sim (d/e)^d/\sqrt{2\pi d}$ and $\ln d! = d \ln d + O(1)$,

$$H'_d = \frac{1}{d}\sum_{k=1}^{d-1} H_k = \frac{1}{d}\sum_{k=1}^{d-1} \ln k + O(1) = \frac{1}{d}\ln d! + O(1) = \ln d + O(1).$$

Hence,

$$\begin{aligned}
\lambda &= H_d = \ln d + O(1),\\
\mu &= d - \lambda = d - \ln d + O(1),\\
\nu &= H'_d + 1 = \ln d + O(1),\\
\xi &= \frac{d+1}{2} - \nu = \frac{d}{2} - \ln d + O(1),\\
\lambda + \xi &= \frac{d}{2} + O(1),\\
\mu\nu - \lambda\xi &= \frac{d \ln d}{2} + O(d),\\
(\lambda + \xi)^2 + 4(\mu\nu - \lambda\xi) &= \left(\frac{d}{2}\right)^2 + 2d \ln d + O(d),\\
\sqrt{(\lambda + \xi)^2 + 4(\mu\nu - \lambda\xi)} &= \frac{d}{2} + 2 \ln d + O(1),
\end{aligned}$$

and finally

$$B_d = \frac{\lambda + \xi + \sqrt{(\lambda + \xi)^2 + 4(\mu\nu - \lambda\xi)}}{2} = \frac{d}{2} + \ln d + O(1).^3 \tag{10}$$

The theorem follows from the fact that $R'(T(d, h)) = f_h$, (7), (9), and (10). □

We now state the main result of this paper.

THEOREM 3. *Let $R(T(d, h))$ be the expected number of leaves evaluated by randomized α-β search for evaluating a uniform tree $T(d, h)$ of degree d and height h. Then for any $T(d, h)$,*

$$R(T(d, h)) = O((B_d)^h)$$

such that

$$\frac{B_d}{B_d^*} = 1 + O\left(\frac{\ln d}{d}\right),$$

where $B_d^* = \left(d - 1 + \sqrt{d^2 + 14d + 1}\right)/4 = d/2 + O(1)$ is the optimal branching factor of randomized algorithms for evaluating uniform game trees of degree d.

Proof. The proof is immediate from Theorem 2 and Corollary 2. □

We remark that by Theorem 3, randomized α-β search could be worse than an optimal randomized algorithm by a factor of $d^{O(h/d)}$ for evaluating uniform game trees of degree d and height h with a total of d^h leaves, as $B_d/B_d^* = 1 + O(\ln d/d) \sim e^{O(\ln d/d)} = d^{O(1/d)}$.

Finally, we show that the upper bound given in Theorem 2 on randomized shallow α-β search is tight. This is done by constructing an instance on which the upper bound is met. This amounts to a construction of an instance that makes the inequalities associated with f_n and g_n into equalities in the proof of Proposition 5. Let $T^*(d, h)$ be an instance of uniform game tree of degree d and height h formed by the following top-down construction. Suppose that we construct the subtree rooted at v of value val(v) that has d children $u_1, u_2, \ldots, u_d$ in the left-to-right order. We set the values of $u_1, u_2, \ldots, u_d$ by the following rules.

[3] A more precise expression is $B_d = \frac{d}{2} + \ln d + \frac{1}{2} + \gamma + O(d^{-1})$ where γ is the Euler's constant.

(i) v is a MAX-node. If v is the root or the first child of its parent, set $\text{val}(u_1) = \text{val}(v) > \text{val}(u_2) > \cdots > \text{val}(u_d)$; otherwise, let w be the parent of v and set $\text{val}(u_1) = \text{val}(v) > \text{val}(w) > \text{val}(u_2) > \cdots > \text{val}(u_d)$, where $\text{val}(v) > \text{val}(w)$ is guaranteed by the top-down construction.

(ii) v is a MIN-node. If v is the first child of its parent, set $\text{val}(u_1) = \text{val}(v) < \text{val}(u_2) < \cdots < \text{val}(u_d)$; otherwise, let w be the parent of v and set $\text{val}(u_1) = \text{val}(v) < \text{val}(w) < \text{val}(u_2) < \cdots < \text{val}(u_d)$ where $\text{val}(v) < \text{val}(w)$ is guaranteed by the top-down construction.

THEOREM 4. *Let $R'(T^*(d, h))$ be the expected number of leaves evaluated by randomized shallow α-β search for evaluating $T^*(d, h)$. Then $R'(T^*(d, h)) = \Theta((B_d)^h)$.*

Proof. The construction of $T^*(d, h)$ gives precisely the tree structure required to make the inequalities associated with f_n and g_n in the proof of Proposition 5 into equalities. The recurrence relations of (3) and (4) can be stated with equalities, and consequently $F_n = f_n$ in Theorem 2. Thus $R'(T^*(d, h)) = f_n = F_n = \Theta((B_d)^h)$ by Theorem 2. □

5. Conclusion. We have shown that the branching factor of randomized α-β search is asymptotically optimal for evaluating uniform game trees as the degree of tree increases. This conclusion is reached through an indirect analysis by showing that the branching factor of randomized shallow α-β search, a weaker version of randomized α-β search using only shallow cutoffs, is asymptotically optimal as the degree of tree increases. It remains a challenging problem to determine the branching factor of randomized α-β search for evaluating uniform trees of a fixed degree.

Acknowledgments. The author is grateful to a referee for suggesting improvements to the proofs, and also thanks Miklos Santha and Stephane Boucheron for their comments.

REFERENCES

[1] G. BAUDET, *On the branching factor of the alpha-beta pruning algorithm*, Artif. Intell., 10 (1978), pp. 173–199.
[2] S. FULLER, J. GASCHNIG, AND J. GILLOGLY, *A analysis of the alpha-beta pruning algorithm*, Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1973.
[3] R. GRAHAM, D. KNUTH, AND O. PATASHNIK, *Concrete Mathematics*, Addison-Wesley, Reading, MA, 1989.
[4] D. KNUTH AND R. MOORE, *An analysis of alpha-beta pruning*, Artif. Intell., 6 (1975), pp. 293–326.
[5] J. PEARL, *Asymptotic properties of minimax trees and game-searching procedures*, Artif. Intell., 14 (1980), pp. 113–138.
[6] ———, *The solution for the branching factor of the alpha-beta pruning algorithm and its optimality*, Comm. Assoc. Comput. Mach., 25 (1982), pp. 559–564.
[7] ———, *Heuristics*, Addison-Wesley, Reading, MA, 1984.
[8] M. SAKS AND A. WIGDERSON, *Probabilistic boolean decision trees and the complexity of evaluating game trees*, in 27th IEEE Annual Symposium on Foundations of Computer Science, 1986, pp. 29–38.
[9] M. TARSI, *Optimal search on some game trees*, J. Assoc. Comput. Mach., 30 (1983), pp. 389–396.

SIAM J. COMPUT.
Vol. 24, No. 1, pp. 148–157, February 1995

010

GREEDY PACKET SCHEDULING*

ISRAEL CIDON†, SHAY KUTTEN‡, YISHAY MANSOUR§, AND DAVID PELEG¶

Abstract. Scheduling packets to be forwarded over a link is an important subtask of the routing process in both parallel computing and in communication networks. This paper investigates the simple class of *greedy* scheduling algorithms, namely, algorithms that always forward a packet if they can. It is first proved that for various "natural" classes of routes, the time required to complete the transmission of a set of packets is bounded by the number of packets, k, and the maximal route length, d, for any greedy algorithm (including the arbitrary scheduling policy). Next, tight time bounds of $d + k - 1$ are proved for a specific greedy algorithm on the class of shortest paths in n-vertex networks. Finally, it is shown that when the routes are arbitrary, the time achieved by various "natural" greedy algorithms can be as bad as $\Omega(d\sqrt{k} + k)$, for any k, and even for $d = \Omega(n)$.

Key words. routing, communication networks, parallel computing, shortest paths

AMS subject classifications. 68M10, 68Q22, 94A05, 90B35, 90B12

1. Introduction.

The task of managing the delivery of packets in a distributed communication network is intricate and complex. Consequently, many routing strategies incorporate design choices directed at simplifying the process. One prime example for this type of choice is the decision to create a clear distinction between two subtasks, namely, *route selection* and *packet scheduling*. The first subtask involves selecting for each packet the route it should use from its source to its destination. This selection is done in advance, before the packet actually leaves its source. The second subtask concerns the transmission stage itself, and involves deciding on the schedule by which the different packets are to be forwarded over each edge along their routes. At this stage, the packets are restricted to their predetermined routes, and cannot deviate from them. This paper concentrates on routing strategies adopting this separation, henceforth referred to as *fixed-route strategies*, and in particular on the scheduling subtask.

A second type of design choice, aimed at simplifying the scheduling process considerably, is to make scheduling decisions *locally* and per packet, rather than globally. The scheduling policy is thus restricted to the selection of local rules for managing the queues on outgoing links, namely, resolving the conflicts between the different packets that need to be advanced on the same outgoing edge. Intuitively, a *local* algorithm has the property that the rules used by a vertex in order to schedule awaiting packets rely only on information concerning these packets (typically contained in the packets' headers), such as the identity of the source and destination, the distance traversed by the packet so far, the arrival time at the current vertex, etc. In contrast, a global algorithm can base its decisions on additional global information on the status of the network, such as the current distribution of packets in the network and the routes of these packets.

Although the two design decisions discussed above may not generally lead to a globally optimal algorithm, they are both widely used. In fact, one of the main distributed network

*Received by the editors July 8, 1991; accepted for publication (in revised form) June 21, 1993.

†IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598, and Faculty of Electrical Engineering, The Technion, Haifa 32000, Israel (cidon@ee.technion.ac.il).

‡IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (kutten@watson.ibm.com).

§ Department of Computer Science, Tel Aviv University, Tel Aviv 69978 (mansour@math.tau.ac.il). The research of this author was partially supported by an IBM graduate fellowship.

¶Department of Applied Mathematics and Computer Science, The Weizmann Institute, Rehovot 76100, Israel (peleg@wisdom.weizmann.ac.il). The research of this author was supported in part by an Allon Fellowship, by a Walter and Elise Haas Career Development Award, and by a Bantrell Fellowship. Part of the work was done while the author was visiting the IBM T.J. Watson Research Center.

strategies for packet routing is *virtual circuit* [CGKK], [Mar82], [BG87], which is based on fixing a predetermined *logical circuit* from the given end user to the given destination, and transmitting *all* packets between them on this circuit. Nonetheless, similar considerations apply also for the second common routing strategy, known as *datagram routing* [MRR80].

Fixed-route strategies are employed in communication networks such as Systems Network Architecture [Mar82], Advanced Peer to Peer Network [BGGJP85], and TYMNET [BG87]. In fact all the emerging integrated high-speed networks proposals such as the international Broadband ISDN (ATM) [CCITT90] and the IBM plaNET Gigabit network [CGGG92], are based on fixed routing strategies. As for the scheduling policy, most networks use a combination of first in, first out (FIFO), certain priority parameters, and flow control information, to determine the next packet to be forwarded. All of these mechanisms are "approximately" local (although flow control adds some global flavor). The main reasons for these choices are based on their advantages from an engineering point of view, namely, their simplicity and low complexity (compared to the global approach), which make them suitable for hardware implementation.

Packet scheduling algorithms for fixed-route strategies were studied by Leighton, Maggs, and Rao [LMR88]. Although motivated by routing problems in specific networks realizing parallel machines, their paper studies the problem on networks of arbitrary topology. The first result of [LMR88] is a proof that there exists a schedule that terminates in $O(d + c)$ time, where d is the maximal route length and c is the maximal congestion, i.e., the maximal number of packets that traverse any edge. However, it seems that determining this schedule requires a complex *centralized* computation, relying on global information. The paper provides also some *randomized* distributed protocols for the problem. These protocols are simple, online, and local (in the sense discussed above). The first applies to arbitrary sets of paths and requires $O(c+d \log |V|)$ time. The second protocol applies to the case when the paths are *leveled* with l levels. Informally, a set of paths is *leveled* if the vertices of the network can be partitioned into l levels in such a way that each edge of the paths connects two consecutive levels. The protocol completes the routing in $O(c+l+\log |V|)$ time. Both protocols of [LMR88] assume all packets start at the same time.

In contrast with both types of algorithms considered in [LMR88], in this paper we consider the complexity of *deterministic distributed* algorithms. In fact, we concentrate on a class of very simple on-line scheduling algorithms, termed *greedy* algorithms. A *greedy algorithm* is an algorithm satisfying the property that at each time unit, the set of packets that are forwarded is maximal, i.e., if there are messages waiting to be forwarded on some link then one of these messages is forwarded. Note that this also includes an algorithm that selects the message to be forwarded next on each link arbitrarily from among the waiting messages (or alternatively, allows an adversary to decide which packet will be sent next). The class of greedy policies is very natural [Ko78], and in fact, all packet scheduling policies used in practical packet switching networks of which we are aware fall in this class. It is interesting to note that the efficient algorithm of [LMR88] is not greedy.

In the upcoming sections we present several results concerning the behavior of greedy scheduling algorithms. We first look at some restricted path classes. To begin with, in §3 we show that for a *leveled* set of paths $\mathcal{P}$, the time required for delivering packet p_i by any greedy algorithm is bounded by $d_i + k - 1$, where d_i is the length of the route of packet p_i and k is the total number of packets. Note that in practical networks k might be much larger than c, hence the randomized algorithm of [LMR88] may perform better.

Our result for the leveled case implies the same result for the natural case of *unique subpaths*. A route collection $\mathcal{P}$ obeys the *unique subpaths* property if for every pair of vertices u and v, all the *subpaths* connecting them in any path of $\mathcal{P}$ are identical.

In §4 we consider the class of *shortest paths*. For this class, we present a strategy that guarantees a bound of $d_i + k - 1$ assuming all packets start at the same time. This strategy is based on advancing the packet that has progressed the least so far. A different strategy, based on fixed priorities, was independently proposed in [RVN90], and shown to yield the same bound. In an earlier version of this paper [CKMP90] we conjectured that the same bound is true for any greedy algorithm. This conjecture has recently been resolved in the affirmative [MP91].

We then turn to general route classes. In contrast with the special cases discussed above, we show in §5 that greedy algorithms might behave badly for an *arbitrary* set of paths. This is true even when we consider natural greedy schedulers, like fixed priority, FIFO, or preferring the packet that traversed the minimum (or maximum) distance so far. We show that in such a case the time may be $\Omega(d\sqrt{k} + k)$. These negative results hold even for the case where both $k = \Theta(|V|)$ and $d = \Theta(|V|)$. This strengthens the counter-examples given in [LMR88] for the case of long routes and a large number of packets.

2. Model. We view the communication network as a directed graph, $G = (V, E)$, where an edge (u, v) represents a bidirectional link connecting the processors u and v. We assume synchronous communication, i.e., the system maintains a global clock, characterized by the property that a packet sent at time t is received by time $t + 1$.

Next let us define formally the routing problem and its relevant parameters. The input to the problem is a collection $\mathcal{P}$ of k packets p_i and k associated routes ρ_i, $1 \leq i \leq k$. Packet p_i, marked by an identifier I_i, is originated at vertex A_i, its destination is B_i, and it is transmitted along the route ρ_i. We deal with vertex-simple (or loop-free) routes. The length of the route ρ_i is d_i, and we denote $d(\mathcal{P}) = \max_i\{d_i\}$.

Two packets are said to *collide* at time t if they are currently waiting at the same vertex to be sent over the same link. The scheduling algorithm has to decide at each time t which packet to forward at time t. (Note that the paths are fixed, and hence the algorithm has no choice with respect to the edges that a packet traverses.) Let τ_i^A denote the time at which packet p_i was sent from its originator A_i, and let τ_i^B denote the arrival time of p_i at its destination B_i. Let T_i denote the time elapsing from τ_i^A until τ_i^B, i.e., $T_i = \tau_i^B - \tau_i^A$. The *schedule time* of $\mathcal{P}$ is $T(\mathcal{P}) = \max_i\{T_i\}$.

Some of our results apply only to special path types. Below we characterize these route classes.

A set of paths $\mathcal{P}$ is leveled if there exists an assignment $level : V \to [1, \ldots, |V|]$, such that for each path $\rho = (v_1 \ldots v_l)$, $\text{level}(v_j) = \text{level}(v_{j-1}) + 1$. A directed graph is leveled if there exists an assignment $level$, such that for every directed edge (u, v), $\text{level}(u) + 1 = \text{level}(v)$. In a leveled directed graph, every set $\mathcal{P}$ of routes is leveled.

The path ρ_i is a *shortest path* if its length equals the distance between its endpoints A_i and B_i. A set of paths $\mathcal{P}$ is *shortest* if every path $\rho_i \in \mathcal{P}$ is a shortest path.

A set of paths $\mathcal{P}$ has the *unique subpaths* property if for every pair of vertices u and v, all the *subpaths* connecting them in any path of $\mathcal{P}$ are identical; that is, if both the routes ρ_i and ρ_j go through u and v, then the segments of the paths connecting u and v are identical.

3. Leveled routing. In this section we prove our first result, concerning greedy scheduling on leveled paths.

THEOREM 3.1. *Let $\mathcal{P}$ be a set of k leveled paths. Then for any greedy algorithm used for routing $\mathcal{P}$,*

1. *every packet p_i arrives within $T_i \leq d_i + k - 1$ time units, and*
2. *the algorithm has schedule time $T(\mathcal{P}) \leq d(\mathcal{P}) + k - 1$.*

Proof. For each packet p_i and $t \geq 0$, let $\text{level}(p_i, t)$ denote the number of the level where p_i resides at time t. A level L is said to be *occupied* at time t if there exists a packet p_i such

that level$(p_i, t) = L$. The proof is based on considering, for any $t \geq 0$, the set $\mathcal{L}(t)$ of levels that are occupied at time t. We shall argue that at every time unit there is some progress, in the sense that either the number of occupied levels grows, or the lowest occupied level (the one whose number is the smallest) becomes unoccupied.

For uniformity of presentation, we adopt the convention that at any time $t > \tau_i^B$ (the time p_i reaches its destination), level(p_i, t) is incremented by one. This can be thought of as if the packet continues progressing indefinitely along some path ρ_i' extended from the destination B_i and dedicated to it, and hence never collides afterwards. This does not restrict generality in any way, since such an extension ρ_i' of the packet's route has no influence on the routes of other packets, and the arrival time of the packet is still considered to be τ_i^B, the time it has reached its original destination B_i.

Consider the collection $\mathcal{L}(t)$ of occupied levels at time t. We break this collection into "blocks" of consecutive levels (separated by unoccupied levels). We define the following parameters for each packet p_i:

- $B(p_i, t)$ is the *block* of p_i at time t (i.e., the block containing level(p_i, t)).

Suppose that $B(p_i, t) = \{L, L+1, \ldots, H\}$. Then

- $\min(p_i, t) = L$.
- $\max(p_i, t) = H$.
- $\text{width}(p_i, t) = |B(p_i, t)| - 1 = \max(p_i, t) - \min(p_i, t)$ $(= H - L)$.

Note that the number of occupied levels at any given time t is bounded by the number of packets, $|\mathcal{L}(t)| \leq k$, and therefore the maximum block size satisfies

$$\text{width}(p_i, t) \leq k - 1. \tag{1}$$

CLAIM 3.2. $\max(p_i, t+1) - \max(p_i, t) \geq 1$ *for every* $t \geq 0$.

Proof. Since the algorithm is greedy, we are guaranteed that if the levels $L, L+1, \ldots, H$ are occupied at time t, then the levels $L+1, \ldots, H+1$ are occupied at time $t+1$. Also, $L \leq \text{level}(p_i, t) \leq H$ implies $L \leq \text{level}(p_i, t+1) \leq H+1$, and therefore $L+1, \ldots, H+1 \in B(p_i, t+1)$. This implies that $\max(p_i + 1, t) \geq H + 1$. □

COROLLARY 3.3. $T_i \leq \max(p_i, \tau_i^B) - \max(p_i, \tau_i^A)$. This corollary is complemented by the following claim which bounds the increase in $\max(p_i, t)$ from above.

CLAIM 3.4. $\max(p_i, \tau_i^B) - \max(p_i, \tau_i^A) \leq d_i + k - 1$.

Proof. Consider a packet p_i whose origin A_i is at level $L_A = \text{level}(p_i, \tau_i^A)$ and whose destination B_i is at level $L_B = \text{level}(p_i, \tau_i^B) = L_A + d_i$. Initially, $\max(p_i, \tau_i^A) \geq L_A$. On the other hand, upon arrival at the destination, $\max(p_i, \tau_i^B) = \min(p_i, \tau_i^B) + \text{width}(p_i, \tau_i^B) \leq L_B + \text{width}(p_i, \tau_i^B)$. Hence by (1) we have that $\max(p_i, \tau_i^B) - \max(p_i, \tau_i^A) \leq L_B + k - 1 - L_A = d_i + k - 1$. □

Combining Corollary 3.3 and Claim 3.4, we get $T_i \leq d_i + k - 1$. This completes the proof of part 1 of Theorem 3.1. Part 2 follows immediately from part 1. □

The natural class of paths with the unique subpaths property can be analyzed using the above theorem.

COROLLARY 3.5. *Let $\mathcal{P}$ be a set of k paths satisfying the unique subpaths property. Then for any greedy algorithm used for routing $\mathcal{P}$,*

1. *every packet p_i arrives within $T_i \leq d_i + k - 1$ time units, and*
2. *the algorithm has schedule time $T(\mathcal{P}) \leq d(\mathcal{P}) + k - 1$.*

Proof. We prove that the delay suffered by any packet p_i is no greater than in an execution on a leveled graph (with the same k and d_i). Consider subgraph G_i induced by the route of a particular packet p_i, and consider the subpaths of this single route which are traversed by other packets. Clearly, because of the unique subpath property each subpath is a consecutive segment of the original route, therefore, G_i is leveled. Consider an execution of of the schedule in the original graph $(\mathcal{P})$ and observe the subgraph G_i. Let τ_j^A in G_i be the time at which packet

p_j arrived to G_i (or p_i's route) in the above execution. Note that the schedules of p_i in G_i and in $\mathcal{P}$ are identical. Thus, by part 1 of Theorem 3.1 the delay suffered by p_i in the unique subpaths case is the same as the one in the leveled paths case we have constructed. □

4. Shortest path routing. In this section we consider a scheduling algorithm for the case in which each route ρ_i in the set $\mathcal{P}$ uses a shortest path from its origin to its destination. We shall assume that all packets start at the same time, i.e., $\tau_i^A = 0$ for $1 \leq i \leq k$. For every time $0 \leq t \leq T_i$, let $d_i(t)$ denote the distance traversed by p_i by time t (note that in particular, $d_i(T_i) = d_i$). If p_i and p_j "collide" at time t, the algorithm resolves the collision based on the distance traversed by the packets so far, breaking ties by packet identifiers. Thus the algorithm will prefer p_i iff

$$d_i(t) < d_j(t) \text{ or } (d_i(t) = d_j(t) \text{ and } I_i < I_j).$$

We refer to this algorithm as the *Min Went* algorithm. The rest of this section is devoted to proving the following theorem.

THEOREM 4.1. *If the set of paths $\mathcal{P}$ consists of shortest paths and $\tau_i^A = 0$ for $1 \leq i \leq k$ (i.e., all the packets start at the same time) then the Min Went scheduling algorithm guarantees*

1. *every packet p_i arrives at time $T_i \leq d_i + k - 1$.*
2. *the schedule time is $T(\mathcal{P}) \leq d(\mathcal{P}) + k - 1$.*

We begin the proof by pointing out the following trivial fact regarding the relationship between packets in consecutive collisions.

FACT 4.2. *If p_i and p_j collide twice (at times t_1 and t_2), then the relation between $d_i(t)$ and $d_j(t)$ is the same at both times.*

DEFINITION 4.3. *Given an execution of the algorithm the* collision relation *C is defined as the collection of all triples $\langle p_i, p_j, t\rangle$ such that at time t packets p_i and p_j collide (i.e., they are at the same vertex, waiting for the same edge), and p_i wins the collision resolution and gets to use the edge (at time t).*

Since only one packet can go on a specific edge at a time t we can deduce the following fact.

FACT 4.4. *For every p, t there is at most one triple $\langle p', p, t\rangle$ in C.*

Consider some packet p; without loss of generality we term it p_0. If this packet is never delayed, then $T_0 = d_0$ and we are done. Hence suppose the packet was delayed along its route. We now define a *delay sequence* for p_0. Let t_0 be the last time that packet p_0 was delayed. (Note that such a time exists since the delays are finite; a bound of $T(\mathcal{P}) \leq k \cdot d(\mathcal{P})$ on the scheduling time of any greedy algorithm is trivial.) Namely, there is a triple $\langle p_1, p_0, t_0\rangle$ in C, and there is no such triple for p_0 in later times $t > t_0$. (Recall that by Fact 4.4 there is only one such p_1.) Let t_1 be the last time p_1 was delayed before time t_0. Namely, there is a triple $\langle p_2, p_1, t_1\rangle$ and no such triple for p_1 in any time between t_1 and t_0. Continue the sequence in this way until reaching a packet p_ℓ that was not delayed prior to time $t_{\ell-1}$.

It is convenient to define also $t_{-1} = T_0$ (the arrival time of p_0) and $t_\ell = 0$ (the start time of p_ℓ).

We get a sequence $\mathcal{DS}$ of triples

$$\mathcal{DS} = \langle p_1, p_0, t_0\rangle, \langle p_2, p_1, t_1\rangle, \ldots, \langle p_\ell, p_{\ell-1}, t_{\ell-1}\rangle,$$

where $T_0 = t_{-1} > t_0 > t_1 > \cdots > t_{\ell-1} > t_\ell = \tau_\ell^A = 0$.

LEMMA 4.5. $T_0 \leq d_0 + \ell$.

Proof. By definition of the relation C and the Min Went scheduling, we have the inequalities

$$(X_j) \qquad d_{j+1}(t_j) \leq d_j(t_j) \qquad \text{for } j = 0, 1, \ldots, \ell - 1.$$

For $j = 0, \ldots, \ell$ let θ_j denote the segment of the route ρ_j traversed by p_j between the times t_j (when it "lost" in the collision resolution) and t_{j-1} (when it won), and let $\Delta_j = |\theta_j| = d_j(t_{j-1}) - d_j(t_j)$. Substitute this definition in the inequalities (X_j) to get

$$(Y_j) \qquad d_{j+1}(t_{j+1}) + \Delta_{j+1} \leq d_j(t_j) \qquad \text{for } j = 0, \ldots, \ell - 1.$$

We also have

$$(Y_{-1}) \qquad d_0(t_0) + \Delta_0 = d_0(t_{-1}) = d_0.$$

Summing the inequalities (Y_j) for $j = -1, 0, \ldots, \ell - 1$, and recalling that $d_\ell(t_\ell) = 0$, we get

$$(2) \qquad \Delta_0 + \Delta_1 + \cdots + \Delta_{\ell-1} + \Delta_\ell \leq d_0$$

We also construct a chain of equalities for the times involved in these collisions. Since packet p_j (for $0 \leq j \leq \ell - 1$) was delayed at time t_j but never delayed since that time until time t_{j-1}, we have

$$(Z_j) \qquad t_{j-1} = t_j + 1 + \Delta_j \qquad \text{for } j = 0, \ldots, \ell - 1.$$

We also have

$$(Z_\ell) \qquad t_{\ell-1} = d_\ell(t_{\ell-1}) = \Delta_\ell + t_\ell = \Delta_\ell$$

Combining the equalities (Z_j) for $j = 0, \ldots, \ell$ we get

$$(3) \qquad T_0 = t_{-1} = \Delta_0 + \Delta_1 + \cdots + \Delta_{\ell-1} + \Delta_\ell + \ell.$$

Combining equations (2) and (3), we get that $T_{i_0} \leq d_{i_0} + \ell$, and the lemma follows. □

In order to complete the proof of the theorem, it therefore remains to bound the length of the sequence $\mathcal{DS}$. This is done by proving the following claim.

LEMMA 4.6. *The packets p_j appearing in the triples of the sequence $\mathcal{DS}$ are all distinct.*

Proof. The proof is by a contradiction. Assume that some packet occurs twice in the sequence, for instance $p_m = p_r$ for $m > r$. (See Fig. 1.) By the structure of the sequence, every two consecutive packets are distinct, so necessarily $m \geq r + 2$. This means that the sequence contains a subcycle

$$\langle p_{r+1}, p_r, t_r\rangle, \ \langle p_{r+2}, p_{r+1}, t_{r+1}\rangle, \ldots, \langle p_{m-1}, p_{m-2}, t_{m-2}\rangle, \ \langle p_m, p_{m-1}, t_{m-1}\rangle = \langle p_r, p_{m-1}, t_{m-1}\rangle,$$

where $t_r > t_{r+1} > \cdots > t_{m-1}$, and $m \geq r + 2$.

We argue that among the inequalities (X_j), for $r \leq j \leq m - 1$, at least one of the inequalities is strict. Otherwise, all the collision resolutions in the cycle were made on the basis of packet identities, so $I_r = I_m < I_{m-1} < \cdots < I_{r+1} < I_r$, which is a contradiction. It follows that among the corresponding inequalities (Y_j), for $r \leq j \leq m - 2$, plus (X_{m-1}), at least one is strict. Combine these inequalities in a chain, to get

$$(4) \qquad d_m(t_{m-1}) + \Delta_{m-1} + \cdots + \Delta_{r+1} < d_r(t_r).$$

Finally, let $\bar{\theta}$ denote the segment of the route ρ_r traversed by p_r between the times t_{m-1} (when it won) and t_r (when it lost), and let $\bar{\Delta} = |\bar{\theta}| = d_r(t_r) - d_r(t_{m-1})$. We get that

$$\Delta_{m-1} + \cdots + \Delta_{r+1} < d_r(t_r) - d_r(t_{m-1}) = \bar{\Delta},$$

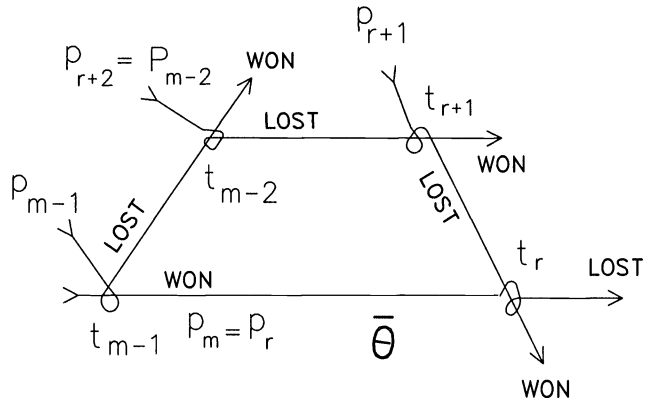


FIG. 1. *Cycle in delay sequence.*

or in other words, the segment $\bar{\theta}$ of ρ_r is not shortest; this is a contradiction to the assumption that all the paths in $\mathcal{P}$ are shortest paths. □

COROLLARY 4.7. *The sequence $\mathcal{DS}$ is of length $\ell \leq k - 1$.*

Combining this corollary with Lemma 4.5 completes the proof of part 1 of the theorem. Part 2 follows immediately.

5. Greedy algorithms in the general case. The purpose of this section is to demonstrate the fact that, unlike the case of leveled routes, for general route classes not every greedy algorithm delivers the messages fast. We start by constructing a directed graph and a set of paths, that will be used as building blocks for our main lower bound construction.

The construction is parameterized by two integers x and y. The constructed directed graph $G_{x,y}$ (see Fig. 2) is composed of a line of $2x + 1$ vertices, denoted by v_0, $v_{\langle 1,\mathtt{in}\rangle}$, $v_{\langle 1,\mathtt{out}\rangle}, \ldots, v_{\langle x,\mathtt{in}\rangle}, v_{\langle x,\mathtt{out}\rangle}$. In addition, each "in" vertex (i.e., $v_{\langle i,\mathtt{in}\rangle}$) is connected to its corresponding "out" vertex (i.e., $v_{\langle i,\mathtt{out}\rangle}$) also via a "detour" path of y vertices (denoted by L^i). Intuitively, the line is the "main" route; the L_i subgraphs are used to "side track" packets and delay them, thus causing them to collide repeatedly.

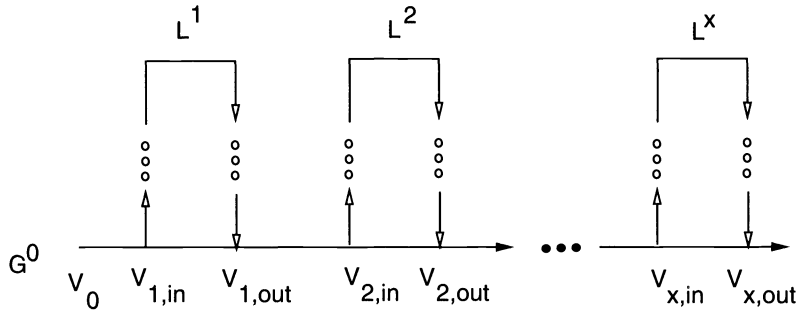


FIG. 2. *The graph $G_{x,y}$.*

A more formal construction follows. Define the graph $G_{x,y} = (V, E)$ to be the union of the subgraphs $G^0, L^1, \ldots, L^x, C$.

The vertices of the subgraph G^0 are

$$V(G^0) = \{1, \ldots, x\} \times \{\mathtt{in}, \mathtt{out}\} \cup \{v_0\},$$

and the edges are

$$E(G^0) = \{(v_{<i,\mathtt{out}>}, v_{<i+1,\mathtt{in}>}) \ : \ 1 \le i \le x-1\} \\ \cup \{(v_{<i,\mathtt{in}>}, v_{<i,\mathtt{out}>}) \ : \ 1 \le i \le x\} \\ \cup \{(v_0, v_{<1,\mathtt{in}>})\}.$$

The graph L is a straight line of y vertices and $y-1$ edges, i.e.,

$$V(L) = \{l_1, \ldots, l_y\} \quad \text{and} \quad E(L) = \{(l_i, l_{i+1}) \ : \ 1 \le i \le y-1\}.$$

For $1 \le i \le x$, the subgraph L^i is a copy of the graph L, with superscript i.

Finally, the edges in C connect the endpoints of each subgraph L^i to the corresponding vertices $v_{<i,\mathtt{in}>}$ and $v_{<i,\mathtt{out}>}$ on the path G^0:

$$C = \{(v_{<i,\mathtt{in}>}, l_1^i) \ : \ 1 \le i \le x\} \cup \{(l_y^i, v_{<i,\mathtt{out}>}) \ : \ 1 \le i \le x\}.$$

Let

$$G_{x,y} = G^0 \cup C \cup \bigcup_{1 \le i \le x} L^i.$$

The paths $\mathcal{P}_{x,y}$ are the following. All the paths start at vertex v_0, and proceed through $v_{<i,\mathtt{in}>}$ and $v_{<i,\mathtt{out}>}$, for $i = 1, \ldots, x$, ending at $v_{<x,\mathtt{out}>}$. There are two types of paths: a "long" path, pl, consisting of $x(y+2)$ edges, and a "short" one, ps, consisting of $2x$ edges. The "short" path ps travels directly through the graph G^0. The "long" path pl takes all the "detours" L^i, i.e., it travels from each "`in`" vertex $v_{<i,\mathtt{in}>}$ to the corresponding "`out`" vertex $v_{<i,\mathtt{out}>}$, via the path L_i. Formally, $ps = E(G^0)$, and

$$pl = \bigcup_{1 \le i \le x} E(L^i) \cup \{(v_{<i,\mathtt{out}>}, v_{<i+1,\mathtt{in}>}) \ : \ 1 \le i \le x-1\} \cup \{(v_0, v_{<1,\mathtt{in}>})\} \cup C.$$

Let us first illustrate the use of this construction by proving a weaker lower bound, and then proceed to our main lower bound. The scheduling policy that we use for the queue scheduling is a fixed priority. This lemma will later be used to prove our main lower bound.

LEMMA 5.1. *For any x and y, there exist a graph $G_{x,y}$ and a collection of $y+1$ paths $\mathcal{P}_{x,y}$, such that there is a packet that traverses a path of length $O(x)$ and requires $\Omega(xy)$ time under the fixed priority queueing policy.*

Proof. Given x and y, construct $G_{x,y}$ as above. The set $\mathcal{P}_{x,y}$ consists of $y+1$ paths, of which y are identical to the "long" path pl, and the remaining one is the "short" path ps. The last packet has the lowest priority. Note that whenever the low priority packet reaches an "`out`" vertex it is delayed y times, once by each other packet. It is also delayed y times in vertex v_0. Therefore, the total delay of this packet is $2x + xy$. □

The above lemma shows that for some packet the delay may be $\Omega(xy)$, even though the number of edges in its route is only $2x$. This lower bound can be strengthened in two respects. Note that in the above construction, some packets have routes of $\Omega(xy)$ edges, which is as high as the lower bound. Also, the number of vertices in the constructed graph is large (i.e., $\Omega(xy)$). In the setting of the next theorem, both the graph size and the route lengths are small relative to the derived lower bound.

THEOREM 5.2. *For every d and k there exist a graph $\mathcal{G}_{d,k}$ and a collection of k paths $\mathcal{P}$ whose schedule time under the fixed priority algorithm is $T(\mathcal{P}) = \Omega(d\sqrt{k})$, and $| V(\mathcal{G}_{d,k}) | = O(d)$.*

Proof. First we create a graph that achieves the delay bound, but has more vertices, and then we show how to reduce the number of vertices. Consider the graph $G_{x,y}$ defined in

Lemma 5.1, with the parameters $x = d/\sqrt{k}$ and $y = \sqrt{k}$. (We assume for simplicity that x and y are integral; if $x < 1$ then $k > d\sqrt{k}$, in which case the claim is trivial.) We take $\sqrt{k}$ copies of $G_{x,y}$, denoted by $G^i_{x,y}$, and connect the ith copy to the $(i+1)$st copy by identifying the last vertex of $G^i_{d,k}$ with the first vertex of $G^{i+1}_{d,k}$ (i.e., $v^i_{<x,\mathrm{out}>} \equiv v^{i+1}_0$).

We partition the packets into $\sqrt{k}$ groups of decreasing priorities; each group is of $\sqrt{k}$ packets. All packets go from $v^1_{<1,\mathrm{in}>}$ to $v^{\sqrt{k}}_{<\sqrt{k},\mathrm{out}>}$. The paths of all the packets within the same group are identical. The ith group traverses in G^j, $j \neq i$, the shortest path from the first vertex to the last vertex (i.e., ps). In G^i the packets traverse all the loops L^j (i.e., use pl). Note that all the paths are of the same length, which is $O(d)$.

Note that the packets in the ith group delay the packets in groups $j > i$ by $\Omega(d)$ time while traversing G^1 (and a similar thing happens as they traverse G^l for $l = 2, 3, \ldots, i-1$). This means that the last packet reaches its destination after $\Omega(d\sqrt{k})$ time.

The number of vertices in the construction so far is $O(d\sqrt{k})$. In order to reduce the number of vertices to $O(d)$ we modify the way in which we combine the $\sqrt{k}$ copies of $G_{x,y}$. For every i, we collapse the $\sqrt{k}$ subgraphs L^i in the graphs $G^1, \ldots, G^{\sqrt{k}}$ into a single copy, namely, L^i of G^1. See Fig. 3. □

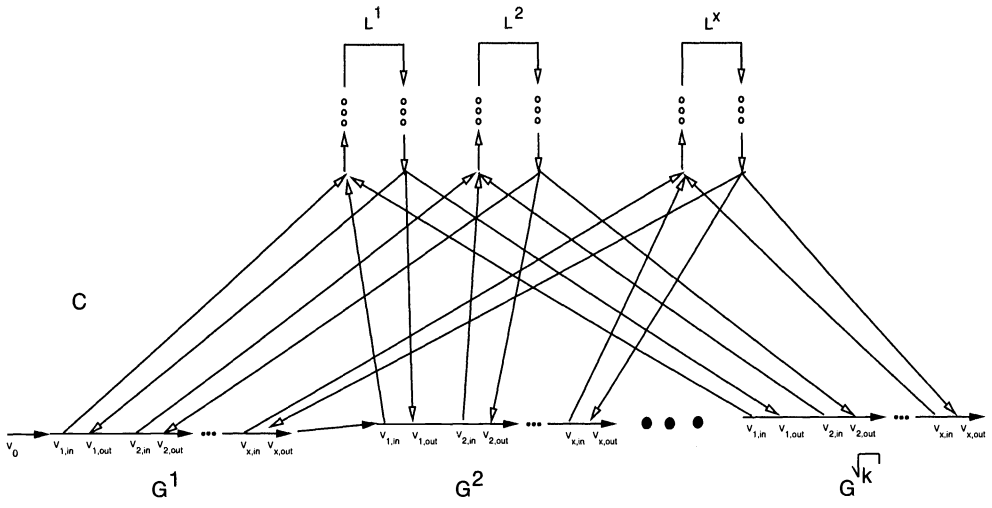


FIG. 3. *The graph* $\mathcal{G}_{d,k}$.

A similar, although somewhat more complicated proof can be constructed for other greedy algorithms, specifically the Min Went policy of §4, the analogous *Max Went* policy, or the *FIFO policy*, namely, the algorithm that resolves a collision between two packets in vertex v by sending the first to have arrived at v. Details can be found in [CKMP90].

Acknowledgment. We thank Amotz Bar-Noy for working with us in the early stages of this research. We are also grateful to the anonymous reviewers for their excellent comments and for simplifying some of the proofs, notably in §3.

REFERENCES

[BGGJP85] A. E. BARATZ, J.P. GRAY, P. E. GREEN, J. M. JAFFE, AND D. P. POZENSKI, *SNA networks of small systems*, IEEE Trans. Comm., sac-3 (1985), pp. 416–426.

[BG87] D. BERTSEKAS AND R. GALLAGER, *Data Networks*, Prentice Hall, Englewood Cliffs, NJ, 1987.

[CCITT90] CCITT SG XVIII, *Draft Recommendation I.121: Broadband Aspects of ISDN*, Geneva, January 1990.

[CGGG92] I. CIDON, I. S. GOPAL, P. M. GOPAL, R. GUERIN, J. JANNIELLO, AND M. KAPLAN, *The plaNET/ORBIT High Speed Network*, IBM Research Technical Report, RC18270, August 1992.

[CKMP90] I. CIDON, S. KUTTEN, Y. MANSOUR, AND D. PELEG, *Greedy packet scheduling*, in Proc. 4th WDAG, Bari, Italy, September, 1990, Lecture Notes in Computer Science, Vol. 486, Springer-Verlag, Berlin, 1991, pp. 169–184.

[CGK88] I. CIDON, I. S. GOPAL, AND S. KUTTEN, *New models and algorithms for future networks*, in Proc. 7th Annual ACM Symp. on Principles of Distributed Computing, Toronto, Canada, August 1988, pp. 74–89.

[CGKK] I. CIDON, I. S. GOPAL, M. KAPLAN, AND S. KUTTEN, *Distributed control for fast networks*, IEEE Trans. Commun., to appear.

[Ko78] H. KOBAYASHI, *Modeling and Analysis*, Addison-Wesley, New York, 1978.

[LMR88] T. LEIGHTON, B. MAGGS, AND S. RAO, *Universal packet routing algorithms*, in Proc. 29th IEEE Symp. on Foundations of Computer Science, White Plains, NY, October 1988, pp. 256–269.

[MP91] Y. MANSOUR AND B. PATT-SHAMIR, *Greedy packet scheduling on shortest paths*, in Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing, August 1991.

[Mar82] J. MARTIN, *SNA: IBM's Networking Solution*, Prentice Hall, Englewood Cliffs, NJ, 1982.

[MRR80] J. MCQUILLAN, I. RICHER, AND E. C. ROSEN, *The new routing algorithm for the ARPANET*, IEEE Trans. Comm., com-28 (1980), pp. 711–719.

[RVN90] P. I. RIVERA-VEGA, R. VARADARAJAN, AND S. B. NAVATHE, *The file redistribution scheduling problem*, Sixth International Conference on Data Eng. Conf., Los Angeles, CA, February 1990, IEEE Computer Society Press, pp. 166–173.

SIAM J. COMPUT.
Vol. 24, No. 1, pp. 158–183, February 1995

011

OPTIMAL FILE SHARING IN DISTRIBUTED NETWORKS*

MONI NAOR† AND RON M. ROTH‡

Abstract. The following file distribution problem is considered: Given a network of processors represented by an undirected graph $G = (V, E)$ and a file size k, an arbitrary file $\mathbf{w}$ of k bits is to be distributed among all nodes of G. To this end, each node is assigned a memory device such that by accessing the memory of its own and of its adjacent nodes, the node can reconstruct the contents of $\mathbf{w}$. The objective is to minimize the total size of memory in the network. This paper presents a file distribution scheme which realizes this objective for $k \gg \log \Delta_G$, where Δ_G stands for the maximum degree in G: For this range of k, the total memory size required by the suggested scheme approaches an integer programming lower bound on that size. The scheme is also constructive in the sense that given G and k, the memory size at each node in G, as well as the mapping of any file $\mathbf{w}$ into the node memory devices, can be computed in time complexity which is polynomial in k and $|V|$. Furthermore, each node can reconstruct the contents of such a file $\mathbf{w}$ in $O(k^2)$ bit operations. Finally, it is shown that the requirement of k being much larger than $\log \Delta_G$ is necessary in order to have total memory size close to the integer programming lower bound.

Key words. derandomization, distributed networks, file assignment, integer programming, linear codes, linear programming, probabilistic algorithms, resource sharing, set cover

AMS subject classifications. 68P20, 68M10, 68Q20, 68R99, 94B05

1. Introduction.

Consider the following file distribution problem: A network of processors is represented by an undirected graph G. An arbitrary file $\mathbf{w}$ of a prescribed size k (measured, say, in bits) is to be distributed among all nodes of G. We are to assign memory devices to the nodes of G such that by accessing the memory of its own and of its adjacent nodes, each node can reconstruct the contents of $\mathbf{w}$. Given G and k, the objective is to find a *static* memory allocation to the nodes of G, independent of $\mathbf{w}$, as to minimize the total size of memory in the network. Although we do not restrict the file distribution or reconstruction algorithms to be of any particular form, we aim at simple and efficient ones.

The problem of file allocation in a network, i.e., of storing a file in a network so that every processor has "easy" access to the file, has been considered in many variants. (See [4] for a survey.) The specific version of reconstruction from adjacent nodes only has received attention in the form of *file segmentation*, where the task is to partition the file so that for each node u in the network, the union of the file segments stored at nodes adjacent to u is the complete file [4], [8], [13]. As we shall see, allowing more general reconstruction procedures than simply taking the union of file segments at adjacent nodes can result in a considerable savings of the total amount of memory required: Letting Δ_G denote the maximum degree of any node in G, the memory requirement of the best segmentation scheme can be $\Omega(\log \Delta_G)$ times larger than the optimal requirement in the general scheme; this bound is tight.

We start by deriving linear and integer programming lower bounds on the total size of memory required for any network G and file size k. We then present a simple scheme that attains these bounds for sufficiently large values of k. In this scheme, however, the file size k must be, in some cases, much larger than $\Delta_G \log \Delta_G$ in order to approach the abovementioned lower bounds. We regard this as a great disadvantage for two reasons: Such a scheme may turn out to be efficient only for large files and, even then, it requires addressing large units of

*Received by the editors September 24, 1992; accepted for publication (in revised form) October 28, 1993. This work was presented in part at the 32nd Institute for Electrical and Electronics Engineers (Inc.) Symposium on Foundations of Computer Science, Puerto Rico, October, 1991.

†Department of Applied Mathematics and Computer Science, Weizmann Institute, Rehovot 76100, Israel (naor@wisdom.weizmann.ac.il). Part of this work was done while the author was with the IBM Research Division, Almaden Research Center, San Jose, California.

‡Computer Science Department, Technion, Haifa 32000, Israel (ronny@cs.technion.ac.il). Part of this work was done while the author was visiting IBM Research Division, Almaden Research Center, San Jose, California.

stored data each time a node accesses the file. Thus we devote considerable attention to the problem of finding a scheme that is close to the linear and integer programming bounds with a file size that is as small as possible.

Our main result is that the critical file size above which the linear or integer programming bounds can be approached is of the order of $\log \Delta_G$: We present a file distribution scheme for any network G and file size k of a total memory size that is within a multiplicative factor of $1 + \epsilon(G, k)$ from the linear programming bound, where $\epsilon(G, k)$ stands for a term which approaches zero as $k/\log \Delta_G$ increases. On the other hand, we present an inifinite sequence of network file-size pairs $\{(G_l, k_l)\}_{l=0}^{\infty}$ such that $k_l \geq \log \Delta_{G_l}$, and yet any file distribution scheme, when applied to a pair (G_l, k_l), requires memory size which is $1 + \delta(G_l, k_l)$ times larger than the integer (or linear) lower bound, with $\liminf_{l\to\infty} \delta(G_l, k_l) \geq \frac{1}{4}$. This proves that a file size of the order of $\log \Delta_G$ is indeed a critical point.

The rest of the paper is organized as follows. In §2 we provide the necessary background and definitions. In §3 we describe the linear and integer programming lower bounds and prove that the linear programming lower bound can be approached for large file sizes k. In §4 we prove our main result, namely, we present a file distribution scheme that approaches the linear programming bound as the ratio $k/\log \Delta_G$ increases. Finally, in §5 we exhibit the fact that a file size of $\log \Delta_G$ is a critical point, below which there exists infinite families of networks for which the linear and integer programming lower bounds cannot be attained.

2. Background and definitions. Throughout this paper we assume the underlying network to be presented by an undirected graph $G = (V, E)$, with a set of nodes $V = V_G$ and a set of edges $E = E_G$ such that

(i) G does not have parallel edges.

(ii) Each node contains a self loop. This stands for the fact that each node can access its own memory.

An undirected graph satisfying conditions (i) and (ii) will be referred to as a *network graph.*

Two nodes u and v in a network graph $G = (V, E)$ are adjacent if there is an edge in G connecting u and v. The adjacency matrix of a network graph $G = (V, E)$ is the $|V| \times |V|$ matrix $A_G = [a_{u,v}]_{u,v\in V}$, where $a_{u,v} = 1$ when u and v are adjacent, and $a_{u,v} = 0$ otherwise. Note that by definition of a network graph, every node $u \in V$ is adjacent to itself and thus $a_{u,u} = 1$.

For every $u \in V$, let $\Gamma(u)$ be the set of nodes that are adjacent to u in G. The degree of u is denoted by $\Delta(u) \triangleq |\Gamma(u)|$, and the maximum degree in G is denoted by $\Delta_G \triangleq \max_{u\in V} \Delta(u)$.

Two real vectors $\mathbf{y} = [y_i]_i$ and $\mathbf{z} = [z_i]_i$ are said to satisfy the relation $\mathbf{y} \geq \mathbf{z}$ if $y_i \geq z_i$ for all i. The scalar product $\mathbf{y} \cdot \mathbf{z}$ of these vectors is defined, as usual, by $\sum_i y_i z_i$. A real vector $\mathbf{y}$ is called nonnegative if $\mathbf{y} \geq \mathbf{0}$, where $\mathbf{0}$ denotes the all-zero vector. By the *norm* of a nonnegative vector $\mathbf{y}$ we mean the L_1-norm $\|\mathbf{y}\| \triangleq \mathbf{y} \cdot \mathbf{1}$, where $\mathbf{1}$ denotes the all-one vector.

Given a network graph $G = (V, E)$ and a positive integer k, a *file distribution protocol for* (G, k) is, intuitively, a procedure for allocating memory devices to the nodes of G and to map an arbitrary file $\mathbf{w}$ of size k into these memory devices such that each node u can reconstruct $\mathbf{w}$ by reading the memory contents at nodes adjacent to u.

More precisely, let $F_2 \triangleq GF(2)$, let $G = (V, E)$ be a network graph, and let k be a positive integer. For $u \in V$ and a real vector $\mathbf{z} = [z_u]_{u\in V}$ denote by $(A_G\mathbf{z})_u$ the uth entry[1] of $A_G\mathbf{z}$; this entry is equal to $\sum_{v\in\Gamma(u)} z_v$. A file distribution protocol χ for (G, k) is a list $(\mathbf{x}; [\mathcal{E}_u]_{u\in V}; [\mathcal{D}_u]_{u\in V})$, consisting of the following:

[1] As we have not defined any order on the set of nodes V, the order of entries in vectors such as $\mathbf{z}$ can be fixed arbitrarily. The same applies to rows and columns of the adjacency matrix A_G, or to subvectors such as $[z_v]_{v\in\Gamma(u)}$.

- *Memory allocation*, which is a nonnegative integer vector $\mathbf{x} = [x_u]_{u \in V}$; the entry x_u denotes the size of memory (in bits) assigned to node u;
- *encoding mappings:*

$$\mathcal{E}_u : F_2^k \to F_2^{x_u} \quad \text{for every } u \in V;$$

these mappings define the coding rule of any file $\mathbf{w}$ of size k into the memory devices at the nodes: the contents of the memory at node u is given by $\mathcal{E}_u(\mathbf{w})$;
- *decoding (reconstruction) mappings:*

$$\mathcal{D}_u : F^{(A_G \mathbf{x})_u} \to F_2^k \text{ for every } u \in V.$$

The memory allocation, encoding mappings, and decoding mappings satisfy the requirement

$$\mathcal{D}_u([\mathcal{E}_v(\mathbf{w})]_{v \in \Gamma(u)}) \equiv \mathbf{w}, \qquad \mathbf{w} \in F_2^k. \tag{1}$$

Equation (1) guarantees that each node u is able to reconstruct the value (contents) of any file $\mathbf{w}$ of size k out of the memory contents $\mathcal{E}_v(\mathbf{w})$ at nodes v adjacent to u.

The *memory size* of a file distribution protocol $\chi = (\mathbf{x}; [\mathcal{E}_u]_{u \in V}; [\mathcal{D}_u]_{u \in V})$ for (G, k) is defined as the norm $\|\mathbf{x}\|$ and is denoted $|\chi|$. That is, the memory size of a file distribution protocol is the total number of bits assigned to the nodes. The minimum memory size of any file distribution protocol for (G, k) is denoted by $M(G, k)$.

Example 1. The file segmentation method mentioned in §1 can be described as a file distribution protocol for (G, k) with memory allocation $\mathbf{x} = [x_u]_{u \in V}$ and associated encoding mappings $\mathcal{E}_u : F_2^k \to F_2^{x_u}$ of the form

$$\mathcal{E}_u : [w_1 w_2 \ldots w_k] \mapsto [w_{j(u;1)} w_{j(u;2)} \ldots w_{j(u;x_u)}],$$

where $0 < j(u; 1) < j(u; 2) < \cdots < j(u; x_u) \leq k$. For a node $u \in V$ to be able to reconstruct the original file $\mathbf{w}$, the mappings $\mathcal{E}_v$, $v \in \Gamma(u)$, must be such that every entry w_i of $\mathbf{w}$ appears in at least one $\mathcal{E}_v(\mathbf{w})$. This implies that the set of nodes S_i, which w_i is mapped to under the encoding mappings, must be a dominating set in G; that is, each node $u \in G$ is adjacent to some node in S_i. On the other hand, given a dominating set S in G, we can construct a file segmentation protocol for (G, k) of memory size $k \cdot |S| \leq k \cdot |V|$ (the case $S = V$ corresponds to simply replicating the original file $\mathbf{w}$ into each node in G).

A *file distribution scheme* is a function $(G, k) \mapsto \chi(G, k)$ which maps every network graph G and positive integer k into a file distribution protocol $\chi(G, k)$ for (G, k).

A file distribution scheme $(G, k) \mapsto \chi(G, k) = (\mathbf{x}; [\mathcal{E}_u]_{u \in V}; [\mathcal{D}_u]_{u \in V})$ is *constructive* if

(a) the complexity of computing the memory allocation $\mathbf{x}$ is polynomial in k and $|V|$;

(b) for every $\mathbf{w} \in F_2^k$, the complexity of computing the encoded values $[\mathcal{E}_u(\mathbf{w})]_{u \in V}$ is polynomial in the memory size $\|\mathbf{x}\|$;

(c) for every $u \in V$ and $\mathbf{c} \in F_2^{(A_G \mathbf{x})_u}$, the complexity of reconstructing $\mathbf{w} = \mathcal{D}_u(\mathbf{c})$ out of $\mathbf{c}$ is polynomial in the original file size k.

By computational complexity of a problem we mean the running time of a Turing machine that solves this problem.

Remark 1. In the definition of memory size of file distribution protocols we chose not to count the amount of memory required at each node u to store and run the routines which implement the decoding mappings $\mathcal{D}_u(\cdot)$. The reasoning for neglecting this auxiliary memory is that, in practice, there are a number of files (each, say, of the same size k) that are to be distributed in the network. The file distribution protocol can be implemented independently for each such file, using the *same* program and the same working space to handle all these files.

To this end, we might better think of k as the size of the smallest information unit (e.g., a word, or a record) that is addressed at each access to any file. From a complexity point of view, we would prefer k to be as small as possible. The motivation of this paper can be summarized as finding a constructive file distribution scheme $(G, k) \mapsto \chi(G, k)$, which maintains a ratio of memory size to file size virtually equal to $\lim_{l\to\infty} M(G, l)/l$ for relatively small file sizes k.

Remark 2. One might think of a weaker definition for constructiveness by allowing nonpolynomial precomputation of $\mathbf{x}$ (item (a)) and, possibly, of other data structures which depend on G and k but not on $\mathbf{w}$ (e.g., calculating suitable representations for $\mathcal{E}_u$ and $\mathcal{D}_u$); such schemes may be justified by the assumption that these precomputation steps should be done once for a given network graph G and a file size k. On the other hand, items (b) and (c) in the constructiveness definition involve the complexity of the more frequent occasions when the file is encoded and—even more so—reconstructed. In this paper, however, we aim at finding file distribution schemes which are constructive in the way we have defined, i.e., in the strong sense: satisfying all three requirements (a)–(c).

We end this section by introducing a few terms which will be used in describing the mappings $\mathcal{E}_u$ and $\mathcal{D}_u$ of the proposed file distribution schemes. Let Φ be a finite alphabet of q elements. An (n, K) *code* C over Φ is a nonempty subset of Φ^n of size K; the parameter n is called the *length* of C, and the members of C are referred to as *codewords*. The *minimum distance* of an (n, K) code C over Φ is the minimum integer d such that any two distinct codewords in C differ in at least d coordinates.

Let C be an (n, K) code over Φ and let S be a subset of $\langle n\rangle \triangleq \{1, 2, \ldots, n\}$. We say that C is *separable with respect to* S if every two distinct codewords in C differ in at least one coordinate indexed by S. The next lemma follows directly from the definition of minimum distance.

LEMMA 1. *The minimum distance of an* (n, K) *code* C *over* Φ *is the minimum integer* d *for which* C *is separable with respect to every set* $S \subseteq \langle n\rangle$ *of size* $n - d + 1$.

Let q be a power of a prime. An (n, K) code C over a field $\Phi = GF(q)$ is *linear* if C is a linear subspace of Φ^n; in this case we have $K = q^k$, where k is the dimension of C. A generator matrix B of a linear (n, q^k) code C over Φ is a $k \times n$ matrix B over Φ whose rows span the codewords of C.

For a $k \times n$ matrix B (such as a generator matrix) and a set $S \subseteq \langle n\rangle$, denote by $(B)_S$ the $k \times |S|$ matrix consisting of all columns of B indexed by S. The following lemma is easily verified.

LEMMA 2. *Let* C *be an* (n, q^k) *linear code over a field* Φ, *let* B *be a generator matrix of* C, *and let* S *be a subset of* $\langle n\rangle$. *Then* C *is separable with respect to* S *if and only if* $(B)_S$ *has rank* k.

3. Lower bounds and the statement of the main result. In this section we first derive lower bounds on $M(G, k)$, i.e., on the memory size of any file distribution protocol for (G, k). Then we state our main result (Theorem 2), which establishes the existence of a constructive file distribution scheme $(G, k) \mapsto \chi(G, k)$ that attains these lower bounds whenever $k \gg \log \Delta_G$. As the proof of Theorem 2 is somewhat long, it is deferred to §4. Instead, we present in this section a simple file distribution scheme which attains the lower bounds when $k = \Omega(\Delta_G^2 \log \Delta_G)$.

3.1. Lower bounds. Let $\mathbf{x} = [x_u]_{u\in V}$ be a memory allocation of some file distribution protocol for (G, k). Assigning x_u bits to each node $u \in V$, each node must "see" at least k memory bits at its adjacent nodes, or else (1) would not hold. Therefore, for every $u \in V$ we must have $\sum_{v\in\Gamma(u)} x_v \geq k$ or, in vector notation,

$$A_G\mathbf{x} \geq k \cdot \mathbf{1}.$$

Let $J(G, k)$ denote the minimum value attained by the following integer programming problem:

$$\text{IP}(G,k): \quad \begin{aligned} &J(G,k) = \min \|\mathbf{y}\|, \\ &\text{ranging over all integer } \mathbf{y} \text{ such that} \\ &A_G\mathbf{y} \leq k \cdot \mathbf{1} \quad \text{and} \quad \mathbf{y} \geq \mathbf{0}. \end{aligned}$$

Also, let ρ_G denote the minimum value attained by the following (rational) linear programming problem:

$$\text{LP}(G): \quad \begin{aligned} &\rho_G = \min \|\mathbf{z}\|, \\ &\text{ranging over all rational } \mathbf{z} \text{ such that} \\ &A_G\mathbf{z} \geq \mathbf{1} \quad \text{and} \quad \mathbf{z} \geq \mathbf{0}. \end{aligned} \tag{2}$$

The next theorem follows from the previous definitions, Example 1, and the fact that $J(G, 1)$ is the size of a (smallest) dominating set in G.

THEOREM 1. *For every network graph G and positive integer k,*

$$\rho_G \cdot k \leq J(G,k) \leq M(G,k) \leq k \cdot J(G,1) \leq k \cdot |V|.$$

We call $J(G, k)$ the *integer programming bound*, whereas $\rho_G \cdot k$ is referred to as the *linear programming bound.*

For $k = 1$, Theorem 1 becomes $M(G, 1) = J(G, 1)$. The problem of deciding whether a network graph G has a dominating set of size $\leq s$ is well known to be NP complete [6]. The next corollary immediately follows.

COROLLARY 1. *Given an instance of a network graph G and positive integers k and s, the problem of deciding whether there exists a file distribution protocol for (G, k) of memory size $\leq s$ (i.e., whether $M(G, k) \leq s$) is* NP *hard.*

Note that we do not know whether the decision problem of Corollary 1 is in NP (and therefore, whether it is NP complete) since it is unclear how to verify (1) in polynomial time, even when the encoding and decoding mappings are computable in polynomial time.

Remark 3. A result of Lovász [11] states that $J(G, 1) \leq \rho_G \log \Delta_G$; on the other hand, one can construct an infinite family of network graphs $\{G_l\}_l$ (such as the ones presented in §5) for which $J(G_l, 1) \geq \frac{1}{4}\rho_{G_l} \log_2 \Delta_{G_l}$ (see also [7]). In terms of file segmentation schemes (Example 1) this means that there always exists a file distribution protocol for (G, k) based on segmentation whose memory size, $k \cdot J(G, 1)$, is within a multiplicative factor of $\log_2 \Delta_G$ from the linear programming bound $\rho_G \cdot k$. Yet, on the other hand, there are families of network graphs for which such a multiplicative gap is definitive (up to a constant 4), even when k tends to infinity.

3.2. Statement of the main result. Corollary 1 suggests that it is unlikely that there exists an efficient algorithm for generating a file distribution scheme $(G, k) \mapsto \chi(G, k)$ with $|\chi(G, k)| = M(G, k)$. This directs our objective to finding a constructive file distribution scheme $(G, k) \mapsto \chi(G, k)$ such that $|\chi(G, k)|/(\rho_G \cdot k)$ is close to 1 for values of k as small as possible.

More specifically, we prove the following theorem.

THEOREM 2. *There exists a constructive file distribution scheme* $(G, k) \mapsto \chi(G, k)$ *such that*

$$\frac{|\chi(G,k)|}{\rho_G \cdot k} = 1 + O\left(\max\left\{\frac{\log \Delta_G}{k}; \sqrt{\frac{\log \Delta_G}{k}}\right\}\right). \tag{3}$$

(The maximum in the right-hand side of (3) is determined according to whether k is smaller or larger than $\log \Delta_G$. Also, by Theorem 1, the ratios $|\chi(G,k)|/M(G,k)$, $M(G,k)/J(G,k)$, and $J(G,k)/(\rho_G \cdot k)$ all approach 1 when $k \gg \log \Delta_G$.)

In §4 we prove Theorem 2 by presenting an algorithm for generating a constructive file distribution scheme $(G, k) \mapsto \chi(G, k)$ which satisfies (3); in particular, the computational complexity of the encoding mappings in the resulting scheme (item (b) in the constructiveness requirements) is $O(k \cdot |\chi(G,k)|)$, whereas applying the decoding mapping at each node (item (c)) requires $O(k^2)$ bits operations. Returning to our discussion in Remark 1, the complexity of these mappings suggests that the file size k should be as small as possible, still greater than $\log \Delta_G$. This means that files distributed in the network should be segmented into records of size $k = a \cdot \log \Delta_G$ for some (large) constant a, each record being encoded and decoded independently. Information can be retrieved from the file by reading whole records of size $a \cdot \log \Delta_G$ bits each, requiring $O(a^2 \log^2 \Delta_G)$ bit operations, whereby the ratio between the memory size required in the network and the file size k is at most $1 + O(1/\sqrt{a})$ times that ratio for $k \to \infty$.

Our file distribution algorithm is divided into two major steps.

Step 1. Find a memory allocation $\mathbf{x} = [x_u]_{u \in V}$ for (G, k) by finding an approximate solution to an integer programming problem; the resulting memory size $|\chi(G,k)| = \|\mathbf{x}\|$ will satisfy (3).

Step 2. Construct a set of $k \times x_u$ matrices B_u, $u \in V$, over F_2; these matrices define the encoding mappings $\mathcal{E}_u : F_2^k \to F_2^{x_u}$ by $\mathcal{E}_u : \mathbf{w} \mapsto \mathbf{w}B_u$, $u \in V$. The choice of the matrices B_u, in turn, is such that each $k \times (A_G\mathbf{x})_u$ matrix $[B_v]_{v \in \Gamma(u)}$ is of rank k, thus yielding decoding mappings $\mathcal{D}_u : F_2^{(A_G\mathbf{x})_u} \to F_2^k$, which satisfy (1).

3.3. File distribution scheme for large files. In this section we present a fairly simple constructive file distribution scheme $(G, k) \mapsto \chi(G, k)$, for which

$$\frac{|\chi(G,k)|}{\rho_G \cdot k} = 1 + O\left(\frac{\Delta_G \cdot \log(\Delta_G \cdot k)}{k}\right).$$

Note that this proves Theorem 2 whenever $k = \Omega(\Delta_G^2 \log \Delta_G)$.

Given a network graph $G = (V, E)$ and a positive integer k, we first compute a memory allocation $\mathbf{x} = [x_u]_{u \in V}$ for (G, k) (Step 1 above). Let $\mathbf{z} = [z_u]_{u \in V}$ be an optimal solution to the linear programming problem LP(G) in (2). Such a vector $\mathbf{z}$ can be found in time complexity which is polynomial in $|V|$ (e.g., by using Karmarkar's algorithm [9]). Set $h \triangleq \lceil \log_2(\Delta_G \cdot k) \rceil$ and $l \triangleq \lceil k/h \rceil$, and define the integer vector $\mathbf{y} = [y_u]_{u \in V}$ by

$$y_u \triangleq \min\{l; \lfloor (l + \Delta_G) \cdot z_u \rfloor\}, \qquad u \in V.$$

Clearly, $\|\mathbf{y}\| \le \rho_G \cdot (l + \Delta_G)$; furthermore, since $A_G\mathbf{z} \ge \mathbf{1}$, we also have

$$(A_G\mathbf{y})_u \ge \min\{l; (l + \Delta_G)(A_G\mathbf{z})_u - \Delta_G\} \ge l, \qquad u \in V,$$

i.e., $A_G\mathbf{y} \ge l \cdot \mathbf{1}$. The memory allocation for (G, k) is defined by $\mathbf{x} \triangleq h \cdot \mathbf{y}$, and it is easy to verify that $\|\mathbf{x}\|/(\rho_G \cdot k) = 1 + O((\Delta_G/k)\log(\Delta_G \cdot k))$.

We now turn to defining the encoding and decoding mappings (Step 2). To this end, we first assign $\Delta_G \cdot l$ colors to the nodes of G, with each node u assigned a set C_u of y_u colors, such that $|\bigcup_{v\in\Gamma(u)} C_v| \geq l, u \in V$. In other words, we multicolor the nodes of G in such a way that each node "sees" at lease l colors at its adjacent nodes.

Such a coloring can be obtained in the following greedy manner: Start with $C_u \leftarrow \emptyset$ for every $u \in V$. Call a node u *saturated* if $|\bigcup_{v\in\Gamma(u)} C_v| \geq l$. (Hence, at the beginning all nodes are unsaturated, whereas at the end all of them should become saturated.) Scan each node $u \in V$ once, and, at each visited node u, redefine the set C_u to have y_u distinct colors not contained in sets C_v already assigned to nodes $v \in \Gamma(u')$ for all unsaturated nodes $u' \in \Gamma(u)$.

To verify that such a procedure indeed yields an all-saturated network, we first show that at each step there are enough colors to assign to the current node. Let $\sigma(u)$ denote the number of unsaturated nodes $u' \in \Gamma(u) - \{u\}$ when C_u is being redefined. Recalling that $y_v \leq l$ for every $v \in V$, it is easy to verify that the number of disqualified colors for C_u is at most $\sigma(u) \cdot (l-1) + (\Delta(u) - \sigma(u) - 1) \cdot l \leq \Delta_G l - l \leq \Delta_G l - y_u$. This leaves at least y_u qualified colors to assign to node u. We now claim that each node becomes saturated at some point. For if node u remained unsaturated all along, then the sets C_v, $v \in \Gamma(u)$, had to be disjoint; but in that case we would have

$$\left|\bigcup_{v\in\Gamma(u)} C_v\right| = \sum_{v\in\Gamma(u)} |C_v| = \sum_{v\in\Gamma(u)} y_u = (A_G\mathbf{y})_u \geq l,$$

contradicting the fact that u was unsaturated.

Let $\alpha_1, \alpha_2, \ldots, \alpha_{\Delta_G \cdot l}$ be distinct elements in $\Phi \triangleq GF(2^h)$, each α_j corresponding to some color j (note that $|\Phi| \geq \Delta_G \cdot k \geq \Delta_G \cdot l$). Given a file $\mathbf{w}$ of k bits, we group the entries of $\mathbf{w}$ into h tuples to form the coefficients of a polynomial $w(t)$ of degree $< \lceil k/h \rceil = l$ over Φ. We now compute the values $w_j = w(\alpha_j)$, $1 \leq j \leq \Delta_G \cdot l$, and store at each node $u \in V$ the values w_j, $j \in C_u$, requiring memory allocation of $x_u = h \cdot y_u$ bits. Since each u has access to images w_j of $w(t)$ evaluated at l distinct elements α_j, each node can interpolate the polynomial $w(t)$ and hence reconstruct the file $\mathbf{w}$.

The above encoding procedure can be described also in terms of linear codes (refer to the end of §2). Such a characterization will turn out to be useful in §§4 and 5. Let B_{RS} be an $l \times (\Delta_G l)$ matrix over $\Phi = GF(2^h)$ defined by $(B_{\mathrm{RS}})_{i,j} = \alpha_j^{i-1}$, $1 \leq i \leq l$, $1 \leq j \leq \Delta_G l$. For every node $u \in V$, let C_u be the set of colors assigned to u and let $B_u \triangleq (B_{\mathrm{RS}})_{C_u}$; that is, regarding C_u as a subset of $\{1, 2, \ldots, \Delta_G l\}$, B_u consists of all columns of B_{RS} indexed by C_u. The mappings $\mathcal{E}_u : F_2^k \to F_2^{x_u}$, or rather, $\mathcal{E}_u : \Phi^l \to \Phi^{y_u}$, are defined by $\mathcal{E}_u : \mathbf{w} \mapsto \mathbf{w}B_u$, $u \in V$, $\mathbf{w} \in \Phi^l$. The matrix B_{RS} is known as a generator matrix of a $(\Delta_G l, 2^{hl})$ generalized Reed–Solomon code over Φ [12, Chaps. 10–11]. Note that since all l columns in B_{RS} are linearly independent, every $l \times (A_G\mathbf{y})_u$ matrix $[B_v]_{v\in\Gamma(u)}$ has rank l, allowing each node u to reconstruct $\mathbf{w}$ out of $[\mathbf{w}B_v]_{v\in\Gamma(u)}$.

We remark that Reed–Solomon codes have been extensively applied to some other reconstruction problems in networks, such as Shamir's secret sharing [18] (see also [10] and [14]).

The file distribution scheme described in this section is not satisfactory when the file size k is, say, $O(\Delta_G)$, in which case the ratio $\chi(G,k)/(\rho_G \cdot k)$ might be bounded away from 1. This will be rectified in our next construction, which is presented in §4.

4. Proof of the main result. In this section we present a file distribution scheme which attains the memory size stated in Theorem 2. In §4.1 we present a randomized algorithm for finding a memory allocation by scaling and perturbing a solution to the linear programming problem LP(G) defined in (2). Having found a memory allocation $\mathbf{x}$, we describe in §4.2

a second randomized algorithm for obtaining the encoding and decoding mappings. Both algorithms are then derandomized in §4.3 to obtain a deterministic procedure for computing the file distribution scheme claimed in Theorem 2. In §4.4 we present an alternative proof of the theorem using the Lovász local lemma. In §4.5 we consider a variant of the cost measure used in the rest of the paper: Instead of looking for a near-optimal solution with respect to the total memory requirement of the system, we consider approximating the best solution such that the maximum amount of memory required in any node is close to the minimum feasible. This is done using the techniques of §4.4.

4.1. Step 1. Solving for a memory allocation. The goal of this section is to prove the following theorem. (Hereafter, e stands for the base of natural logarithms.)

THEOREM 3. *Given a network graph G and an integer m, let $\mathbf{z} = [z_u]_{u \in V}$ be a nonnegative real vector satisfying $A_G \mathbf{z} \geq \mathbf{1}$. Then there is a nonnegative integer vector $\mathbf{x}$ satisfying $A_G \mathbf{x} \geq m \cdot \mathbf{1}$ such that*

$$\frac{\|\mathbf{x}\|}{\|m \cdot \mathbf{z}\|} \leq 1 + c \cdot \max\left\{\frac{\log_e \Delta_G}{m}; \sqrt{\frac{\log_e \Delta_G}{m}}\right\} \tag{4}$$

for some absolute constant c.

In fact, we provide also an efficient algorithm to compute the nonnegative integer vector $\mathbf{x} = [x_u]_{u \in V}$ guaranteed by the theorem. The vector $\mathbf{x}$ will serve as the memory allocation of the computed file distribution protocol for an instance (G, k), where we will need to take m slightly larger than k in order to construct the encoding and decoding mappings in §4.2.

Theorem 3 is proved via a "randomized rounding" argument (see [15] and [17]): We first solve the corresponding linear programming problem LP(G) in (2) (say, by Karmarkar's algorithm [9]), and use the rational solution to define a probability measure on integer vectors that are candidates for $\mathbf{x}$. We then show that this probability space contains an integer vector $\mathbf{x}$, which satisfies the conditions of Theorem 3. Furthermore, such a vector can be found by a polynomial-time (randomized) algorithm. Note that if we are interested in a weaker result, where $\log |V|$ replaces $\log \Delta_G$ in Theorem 2 (or in Theorem 3), then a slight modification of Raghavan's lattice approximation method can be applied [15]. However, to prove Theorem 3 as is, we need a so-called "local" technique. One possibility is to use the "method of alteration" (see [19]), where a random integer vector selected from the above probability space is perturbed in a few coordinates so as to satisfy the conditions of the theorem. Another option is to use the Lovász local lemma. Both methods can be used to prove Theorem 3, and both can be made constructive and deterministic: The method of alteration can be used by applying the method of conditional probabilities (see Spencer [19, p. 31] and Raghavan [15]), and the local lemma can be used with Beck's method [2]. We show here the method of alteration, then present a second existence proof using the local lemma in §4.4.

Given a nonnegative real vector $\mathbf{z} = [z_u]_{u \in V}$ and a real number $\ell > 0$, define the vectors $\mathbf{s} = [s_u]_{u \in V}$ and $\mathbf{p} = [p_u]_{u \in V}$ by

$$s_u \triangleq \lfloor \ell \cdot z_u \rfloor \quad \text{and} \quad p_u \triangleq \ell \cdot z_u - s_u, \qquad u \in V; \tag{5}$$

note that $0 \leq p_u < 1$ for every $u \in V$. Let $\mathbf{Y} = [Y_u]_{u \in V}$ be a random vector of independent random variables Y_u over $\{0, 1\}$ such that

$$\text{Prob}\{Y_u = 1\} = p_u, \qquad u \in V, \tag{6}$$

and let $\mathbf{X} = [X_u]_{u \in V}$ be a random vector defined by

$$\mathbf{X} \triangleq \mathbf{s} + \mathbf{Y}. \tag{7}$$

Fix $\mathbf{a}$ to be a real vector in the unit hypercube $[0, 1]^{|V|}$ such that $\mathbf{a} \cdot \mathbf{z} \geq 1$. Since the expectation vector $E(\mathbf{Y})$ is equal to $\mathbf{p}$, we have

$$E(\mathbf{a} \cdot \mathbf{X}) = \mathbf{a} \cdot \mathbf{s} + \mathbf{a} \cdot \mathbf{p} = \ell \cdot \mathbf{a} \cdot \mathbf{z} \geq \ell.$$

In particular, if $\mathbf{z}$ is a rational vector satisfying $A_G \mathbf{z} \geq \mathbf{1}$, then

$$E(A_G \mathbf{X}) \geq \ell \cdot A_G \mathbf{z} \geq \ell \cdot \mathbf{1}.$$

Showing the existence of an instance of $\mathbf{X}$ which can serve as the desired memory allocation $\mathbf{x}$ makes use of the following two propositions. The proofs of these propositions are given in the Appendix, as similar statements can be found also in [15].

Throughout this section, $L\{\beta, \eta\}$ stands for $\max\{\log_e \beta; \sqrt{\eta \cdot \log_e \beta}\}$.

PROPOSITION 1. *Given a nonnegative real vector* $\mathbf{z}$ *and an integer* ℓ, *let* $\mathbf{X} = [X_u]_{u \in V}$ *be defined by* (5)–(7), *let* $\mathbf{a}$ *be a real vector in* $[0, 1]^{|V|}$ *such that* $\mathbf{a} \cdot \mathbf{z} \geq 1$, *and let* m *be a positive integer. There exists a constant* c_1 *such that for every* $\beta \geq 1$,

$$\text{Prob}\{\mathbf{a} \cdot \mathbf{X} < m\} \leq \frac{1}{\beta}$$

whenever $\ell \geq m + c_1 \cdot L\{\beta, m\}$.

PROPOSITION 2. *Given a nonnegative real vector* $\mathbf{z}$ *and an integer* ℓ, *let* $\mathbf{X} = [x_u]_{u \in V}$ *be defined by* (5)–(7) *and let* $\mathbf{a}$ *be a real vector in* $[0, 1]^{|V|}$. *There exists a constant* c_2 *such that for every* $\beta \geq 1$.

$$\text{Prob}\{\mathbf{a} \cdot \mathbf{X} > E(\mathbf{a} \cdot \mathbf{X}) + c_2 \cdot L\{\beta, E(\mathbf{a} \cdot \mathbf{X})\}\} \leq \frac{1}{\beta}.$$

Consider the following algorithm for computing a nonnegative integer vector $\mathbf{x}$ for an instance (G, m).

ALGORITHM 1.

1. *Set* $\beta = \beta_G \triangleq 2\Delta_G^2$ *and* $\ell = m + c_1 \cdot L\{\beta_G, m\}$.
2. *Solve the linear programming problem* LP(G) (*defined by* (2)) *for* $\mathbf{z}$.
3. *Generate an instance of the random vector* $\mathbf{X} = [s_u]_u + [Y_u]_u$ *as in* (5)–(7).
4. *The integer vector* $\mathbf{x} = [x_u]_{u \in V}$ *is given by*

$$x_u \triangleq \begin{cases} s_u + 1 & \text{if there exists } v \in \Gamma(u) \text{ with } (A_G \mathbf{X})_v < m, \\ s_u + Y_u & \text{otherwise.} \end{cases}$$

Theorem 3 is a consequence of the following lemma.

LEMMA 3. *The vector* $\|\mathbf{x}\|$ *obtained by Algorithm* 1 *satisfies inequality* (4) *with probability* $\geq \frac{1}{2} - (1/\beta_G)$.

Proof. Call a node v *deficient* if $(A_G \mathbf{X})_v < m$ for the generated vector $\mathbf{X}$. First note that x_u is either X_u or $X_u + 1$ and that

$$A_G \mathbf{x} \geq m \cdot \mathbf{1}; \tag{8}$$

in fact, for deficient nodes v we have $(A_G \mathbf{x})_v \geq \ell \cdot (A_G \mathbf{z})_v \geq \ell \geq m$.

Now by Proposition 1, for every node $v \in V$,

$$\text{Prob } \{ \text{ node } v \text{ is deficient } \} = \text{Prob } \{(A_G \mathbf{X})_v < m\} \leq \frac{1}{\beta_G}.$$

Hence, for each node $u \in V$,

$$\text{Prob}\{x_u = X_u + 1\} \leq \Delta_G \cdot \text{Prob}\{\text{node } v \text{ is deficient}\} \leq \frac{\Delta_G}{\beta_G} = \frac{1}{2\Delta_G}.$$

Therefore, the expected number of nodes u for which $x_u = X_u + 1$ is at most $(|V|/2\Delta_G)$ and, with probability at least $\frac{1}{2}$, there are no more than $|V|/\Delta_G$ such nodes u. Observing that

$$\|\mathbf{z}\| \geq \frac{1}{\Delta_G}\sum_{u\in V}(A_G\mathbf{z})_u \geq \frac{1}{\Delta_G}\sum_{u\in V}1 = \frac{|V|}{\Delta_G},$$

we thus obtain, with probability $\geq \frac{1}{2}$,

$$\|\mathbf{x}\| \leq \|\mathbf{X}\| + \frac{|V|}{\Delta_G} \leq \|\mathbf{X}\| + \|\mathbf{z}\|. \tag{9}$$

Recalling that $E(\|\mathbf{X}\|) = \ell \cdot \|\mathbf{z}\|$, we apply Proposition 2 with $\mathbf{a} = \mathbf{1}$ to obtain

$$\text{Prob}\{\|\mathbf{X}\| > \ell \cdot \|\mathbf{z}\| + c_2 \cdot L\{\beta_G, \ell \cdot \|\mathbf{z}\|\}\} \leq \frac{1}{\beta_G}. \tag{10}$$

Hence, by (8)–(10) we conclude that with probability $\geq \frac{1}{2} - (1/\beta_G)$, the integer vector $\mathbf{x}$ satisfies both

$$A_G\mathbf{x} \geq m \cdot \mathbf{1} \tag{11}$$

and

$$\begin{aligned}\|\mathbf{x}\| &\leq (\ell+1)\cdot\|\mathbf{z}\| + c_2 \cdot L\{\beta_G, \ell\cdot\|\mathbf{z}\|\}\\ &\leq (\ell+1)\cdot\|\mathbf{z}\| + c_2\cdot\|\mathbf{z}\|\cdot L\{\beta_G,\ell\}.\end{aligned}$$

The last inequality implies

$$\frac{\|\mathbf{x}\|}{\|\ell\cdot\mathbf{z}\|} \leq 1 + \frac{1}{\ell} + c_2\left(\frac{\log_e\beta_G}{\ell} + \sqrt{\frac{\log_e\beta_G}{\ell}}\right), \tag{12}$$

and the lemma now follows by substituting

$$\ell = m\cdot\left(1 + c_2\cdot\max\left\{\frac{\log_e\beta_G}{m}; \sqrt{\frac{\log_e\beta_G}{m}}\right\}\right)$$

and $\beta_G = 2\Delta_G^2$ in (12). □

Note that for $m = k + O(\log\Delta_G)$ we also have

$$\frac{\|\mathbf{x}\|}{\|k\cdot\mathbf{z}\|} = 1 + O\left(\max\left\{\frac{\log\Delta_G}{k}; \sqrt{\frac{\log\Delta_G}{k}}\right\}\right) \tag{13}$$

(compare with the right-hand side of (3)). The vector $\mathbf{x}$, computed for $m = k + O(\log\Delta_G)$, will serve, with a slight modification, as the memory allocation of $\chi(G,k)$. In §4.3 we shall apply the method of conditional probabilities to make Algorithm 1 deterministic.

4.2. Step 2. Defining the encoding mappings. Having found a memory allocation $\mathbf{x}$, we now provide a randomized algorithm for constructing the encoding and decoding mappings. The construction makes use of the following lemma.

LEMMA 4 [12, p. 444]. *Let* $\mathbf{S}$ *denote a random matrix, uniformly distributed over all* $k \times m$ *matrices over* F_2. *Then*

$$\text{Prob}\{\text{rank}(\mathbf{S}) = k\} = \prod_{i=0}^{k-1}(1 - 2^{i-m}) > 1 - 2^{k-m}.$$

Given an instance (G, k), let $\mathbf{x} = [x_u]_{u\in V}$ be the nonnegative integer vector obtained by Algorithm 1 for $m = k + 3\lceil \log_2 \Delta_G \rceil + 1$. The following algorithm computes for each node u a matrix B_u to be used for the encoding mappings.

ALGORITHM 2.

1. *For each* $u \in V$, *assign at random a matrix* $\mathbf{Q}_u$ *uniformly distributed over all* $k \times x_u$ *matrices over* F_2.

2. *For each* $u \in V$, *let* $\mathbf{S}_u \triangleq [\mathbf{Q}_v]_{v\in\Gamma(u)}$, *and define the encoding matrix* B_u *by*

$$B_u \triangleq \begin{cases} \mathbf{Q}_u & \text{if rank } (\mathbf{S}_u) = k, \\ I_k & \text{if rank } (\mathbf{S}_u) < k, \end{cases} \tag{14}$$

where I_k *stands for the* $k \times k$ *identity matrix.*

Note that each B_u is a $k \times \hat{x}_u$ binary matrix with

$$\hat{x}_u = \begin{cases} x_u & \text{if rank } (\mathbf{S}_u) = k, \\ k & \text{if rank } (\mathbf{S}_u) < k. \end{cases} \tag{15}$$

The vector $\hat{\mathbf{x}} \triangleq [\hat{x}_u]_{u\in V}$ will serve as the (final) memory allocation for $\chi(G, k)$. As we show later on in this section, the excess of $\|\hat{\mathbf{x}}\|$ over $\|\mathbf{x}\|$, if any, is small enough to let (13) hold also with respect to the memory allocation $\hat{\mathbf{x}}$. This will establish the memory size claimed in Theorem 2. The associated encoding mappings $\mathcal{E}_u : F_2^k \to F_2^{\hat{x}_u}$ are given by $\mathcal{E}_u : \mathbf{w} \mapsto \mathbf{w}B_u$, $u \in V$, and the overall process of encoding $\mathbf{w}$ into $[\mathcal{E}_u(\mathbf{w})]_{u\in V}$ requires $O(k \cdot \|\hat{\mathbf{x}}\|)$ multiplications and additions over F_2.

Recalling the definitions in §2, note that for each node u, the $k \times \|\hat{\mathbf{x}}\|$ matrix $B \triangleq [B_v]_{v\in V}$ is separable with respect to the set $\Gamma(u)$; that is, the rank of $(B)_{\Gamma(u)} = [B_v]_{v\in\Gamma(u)}$ is k. Therefore, each node u, knowing the values $[\mathcal{E}_v(\mathbf{w})]_{v\in\Gamma(u)} = [\mathbf{w}B_u]_{v\in\Gamma(u)}$, is able to reconstruct the file $\mathbf{w}$. To this end, node u has to process only k fixed coordinates of $\mathbf{w}(B)_{\Gamma(u)}$, namely, k coordinates which correspond to k linearly independent columns of $(B)_{\Gamma(u)}$. Let such a set of coordinates be indexed by the set T_u, $u \in V$. Assuming a "hard-wired" connection between node u and the k entries of $\mathbf{w}(B)_{\Gamma(u)}$ indexed by T_u, the decoding process at u sums up to multiplying the vector $\mathbf{w}(B)_{T_u} \in F_2^k$ by the inverse of $(B)_{T_u}$. Hence, the mappings $\mathcal{D}_u$, $u \in V$, are given by $\mathcal{D}_u(\mathbf{c}) = (\mathbf{c})_{T_u}((B)_{T_u})^{-1}$ for every $\mathbf{c} \in F^{(A_G\hat{\mathbf{x}})_u}$. The decoding process at each node thus requires $O(k^2)$ multiplications and additions over F_2. Note that in those cases where we set B_u in (14) to be the identity matrix, the decoding process is trivial, since the whole file is written at node u.

We now turn to estimating the memory size $\hat{\mathbf{x}}$. First note that for every node u, the matrix $\mathbf{S}_u$ is uniformly distributed over all $k \times (A_G\mathbf{x})_u$ matrices over F_2. Recalling that, by construction, $(A_G\mathbf{x})_u \geq m = k + 3\lceil \log_2 \Delta_G \rceil + 1$, we have, by Lemma 4,

$$\text{Prob}\{\text{rank } (\mathbf{S}_u) < k\} < 2^{k-m} \leq \frac{1}{2\Delta_G^3}.$$

Hence, the expected number of nodes for which $\hat{x}_u > x_u$ in (15) is at most $|V|/(2\Delta_G^3)$. Therefore, with probability at least $\frac{1}{2}$, there are no more than $|V|/\Delta_G^3$ nodes u whose memory allocation x_u has been increased to $\hat{x}_u = k$. Since $|V|/\Delta_G^3 \le \|\mathbf{z}\|/\Delta_G^2$, the total memory-size increase in (15) is bounded from above by $(k/\Delta_G^2)\|\mathbf{z}\|$. Hence, by (13),

$$\frac{\|\hat{\mathbf{x}}\|}{\|k \cdot \mathbf{z}\|} \le \frac{\|\mathbf{x}\| + (k/\Delta_G^2)\|\mathbf{z}\|}{\|k \cdot \mathbf{z}\|} = 1 + O\left(\max\left\{\frac{\log \Delta_G}{k}; \sqrt{\frac{\log \Delta_G}{k}}\right\}\right)$$

whenever $k = O(\Delta_G^2 \log \Delta_G)$. Recall that the construction of §3.3 covers Theorem 2 for larger values of k.

In §4.3 we apply the method of conditional probabilities (see [19, p. 31] and [15]) in order to make the computation of the matrices B_u deterministic.

Remark 4. It is worthwhile comparing the file distribution scheme described in §§4.1 and 4.2 with the scheme of §3.3, modified to employ Algorithm 1 on $(G, \lceil k/h \rceil)$, $h = \lceil \log_2(\Delta_G \cdot k) \rceil$, to solve for the memory allocation there. It can be verified that the resulting file distribution scheme is slightly worse than the one obtained here: Every term $\log \Delta_G$ in (3) should be changed to $\log(\Delta_G \cdot k) \log \Delta_G$. In particular, this method has critical file size of $\log^2 \Delta_G$.

4.3. A deterministic algorithm. We now show how to make Algorithms 1 and 2 deterministic using the method of conditional probabilities of Spencer [19, p. 31] and Raghavan [15], adapted to conditional expectation values. The idea of the method of conditional probabilities is to search the probability space defined by the random choices. At each iteration the probability space is bisected by setting one of the random variables. Throughout the search we estimate the probability of success, conditional on the choices we have fixed so far. The value of the next random variable is chosen as the one that maximizes the estimator function.

In derandomizing Algorithms 1 and 2 we employ as an estimator the expected value of the size of the allocation. At every step the conditional expectation for both possibilities for the value of the next random variable are computed and the setting that is smaller (thus increasing the probability of success) is chosen. Unlike Raghavan [15], we do not employ a "pessimistic estimator," but rather a conditional expectation estimator which is fairly easy to compute.

We start with derandomizing the computation of the (initial) memory allocation $\mathbf{x}$. Let $\mathbf{z} = [z_u]_u = [s_u + p_u]_u$, $\mathbf{X} = [X_u]_u$, and $\mathbf{x} = [x_u]_u$ be the vectors computed in the course of Algorithm 1. Recall that for every $u \in V$, the entry X_u is a random variable given by $X_u = s_u + Y_u$, with Prob $\{Y_u = 1\} = p_u$. Now,

$$\begin{aligned} E(\|\mathbf{x}\|) &= E(\|\mathbf{X}\|) + \sum_{u \in V} \text{Prob}\{x_u = X_u + 1\} \\ &\le E(\|\mathbf{X}\|) + \Delta_G \cdot \sum_{u \in V} \text{Prob}\{\text{ node } u \text{ is deficient }\} \\ &\triangleq \hat{E}. \end{aligned}$$

We refer to $\hat{E}$ as the *expectation estimator* for $\mathbf{x}$, and we have

$$\begin{aligned} E(\|\mathbf{x}\|) \le \hat{E} &= E(\|\mathbf{X}\|) + \Delta_G \cdot \sum_{u \in V} \text{Prob}\{(A_G\mathbf{X})_u < m\} \\ &= \ell \cdot \|\mathbf{z}\| + \Delta_G \cdot \sum_{u \in V} \text{Prob}\left\{\sum_{v \in \Gamma(u)} X_v < m\right\} \\ &\le \ell \cdot \|\mathbf{z}\| + \frac{|V|}{2\Delta_G} \le (\ell + \tfrac{1}{2}) \cdot \|\mathbf{z}\|. \end{aligned}$$

Comparing the last inequality with (12), it would suffice if we found a memory allocation whose size is at most $\hat{E}$. Note that $\hat{E}$ can be computed efficiently by calculating the expressions Prob $\{\sum_{v\in W_i} X_v < j\}$ for subsets W_i of $\Gamma(u)$ consisting of the first i nodes in $\Gamma(u)$ for $i = 1, 2, \ldots, \Delta(u) = |\Gamma(u)|$, and for $\sum_{u\in W} s_u \le j \le m$. Such a computation can be carried out efficiently by dynamic programming.

Let Y_1 denote the first entry of $\mathbf{Y} = \mathbf{X} - \mathbf{s}$ and define the conditional expectation estimators by

$$\hat{E}_b \triangleq E(\|\mathbf{X}\||Y_1 = b) + \Delta_G \cdot \sum_{u\in V} \text{Prob}\left\{\sum_{v\in\Gamma(u)} X_v < m|Y_1 = b\right\}, \qquad b = 0, 1.$$

Indeed, we have $E(\|\mathbf{x}\||Y_1 = b) \le \hat{E}_b$; furthermore, the two conditional expectation estimators $\hat{E}_0$ and $\hat{E}_1$ have $\hat{E}$ as a convex combination and, therefore, one of them must be bounded from above by $\hat{E}$. We set the entry Y_1 to the bit $y_1 = b$ for which $\hat{E}_b$ is the smallest. Note that, like $\hat{E}$, the conditional expectation estimators can be efficiently computed.

Having determined the first entry in $\mathbf{Y}$, we now reiterate this process with the second entry, Y_2, now involving the conditional expectation estimators $\hat{E}_{y_1,0}$ and $\hat{E}_{y_1,1}$. Continuing this way with subsequent entries of $\mathbf{Y}$, we end up with a nondecreasing sequence of conditional expectation estimators

$$\begin{aligned}(\ell + \tfrac{1}{2}) \cdot \|\mathbf{z}\| \ge \hat{E} \ge \hat{E}_{y_1} \ge \hat{E}_{y_1,y_2} \ge \cdots \ge \hat{E}_{y_1,y_2,\ldots,y_{|V|}} \\ \ge E(\|\mathbf{x}\||y_1 = y_1, Y_2 = y_2, \ldots, Y_{|V|} = y_{|V|}),\end{aligned}$$

thus determining the whole vector $\mathbf{Y}$, and therefore the vectors $\mathbf{X}$ and $\mathbf{x}$, the latter having memory size $\le (\ell + \frac{1}{2}) \cdot \|\mathbf{z}\|$.

We now turn to making the computation of the encoding mappings deterministic. Recall that Algorithm 2 first assigns a random $k \times x_u$ matrix $\mathbf{Q}_u$ to each node u. We may regard this assignment as an $\|\mathbf{x}\|$-step procedure, where at the nth step a random column of F_2^k is added to a node v with less than x_v already-assigned columns. Denote by $\mathbf{Q}_{u;n}$ the (partial) matrix at node $u \in V$ after the nth step. The assignment of the random matrices $\mathbf{Q}_u$ to the nodes of the network can thus be described as a random process $\{\mathbf{U}_n\}_{n=1}^{\|\mathbf{x}\|}$, where $\mathbf{U}_n = \{\mathbf{Q}_{u;n}\}_{u\in V}$ is a random column configuration denoting the contents of each node after adding the nth column to the network graph. We shall use the notation $\mathbf{U}_0$ for the initial column configuration where no columns have been assigned yet to any node.

Let $\mathbf{S}_u$ denote the random matrix $[\mathbf{Q}_v]_{v\in\Gamma(u)}$ (as in Algorithm 2) and let R be the number of nodes u for which rank $(\mathbf{S}_u) < k$. Recall that Algorithm 2 was based on the inequality

$$E(R) \triangleq E(R|\mathbf{U}_0) < \frac{|V|}{2\Delta_G},$$

which then allowed us to give a probabilistic estimate of $2E(R) < \frac{|V|}{\Delta_G}$ for the number of nodes u that required the replacement of $\mathbf{Q}_u$ by I_k. Instead, we compute here a sequence of column configurations $U_n = \{Q_{u;n}\}_{u\in V}$, $n = 1, 2, \ldots, \|\mathbf{x}\|$, such that

$$E(R|\mathbf{U}_n = U_n) \le E(R|\mathbf{U}_{n-1} = U_{n-1}); \tag{16}$$

in particular, we will have

$$E(R|\mathbf{U}_{\|\mathbf{x}\|} = U_{\|\mathbf{x}\|}) < \frac{|V|}{2\Delta_G},$$

i.e., the number of nodes u for which B_u is set to I_k in (14) is guaranteed to be less than $|V|/(2\Delta_G)$.

In order to attain the inequality chain (16) we proceed as follows: Let U_0 be the empty column configuration and assume, by induction, that the column configuration U_{n-1} has been determined for some $n \geq 1$. Let v be a node which has been assigned less than x_v columns in U_{n-1}. We now determine the column which will be added to v to obtain U_n. This is done in a manner similar to the process described before derandomizing Algorithm 1: Set the first entry, b_1, of the added column to be 0, assume the other entries to be random bits, and compute the expected value, E_0, of R conditioned on $\mathbf{U}_{n-1} = U_{n-1}$ and on $b_1 = 0$. Now repeat the process with b_1 being set to 1, resulting in a conditional expected value E_1 of R. Since the two conditional expected values E_0 and E_1 average to $E(R|\mathbf{U}_{n-1} = U_{n-1})$, one of them must be at most that average. The first entry b_1 in the column added to v is set to the bit b for which E_b is the smallest. This process is now iterated for the second bit b_2 of the column added to v, resulting in two conditional expected values $E_{b_1,0}$ and $E_{b_1,1}$ of R, the smaller of which determines b_2. Continuing this way, we obtain a sequence of conditional expected values of R,

$$E(R|\mathbf{U}_{n-1} = U_{n-1}) \geq E_{b_1} \geq E_{b_1,b_2} \geq \cdots \geq E_{b_1,b_2,\ldots,b_k},$$

thus determining the entire column added to v. Note that, indeed,

$$E(R|\mathbf{U}_n = U_n) = E_{b_1,b_2,\ldots,b_k} \leq E(R|\mathbf{U}_{n-1} = U_{n-1}),$$

in accordance with (16).

It remains to show how to compute the conditional expected values of R which are used to determine the column configurations U_n. It is easy to verify that for any event $\mathcal{A}$,

$$E(R|\mathcal{A}) = \sum_{u \in V} \text{Prob } \{ \text{ rank } (\mathbf{S}_u) < k|\mathcal{A}\}. \tag{17}$$

Hence, the computation of the conditional expected values of R boils down to the following problem:

Let $\mathbf{S}$ *denote a* $k \times m$ *random matrix over* F_2, *whose first* l *columns, as well as the first* t *entries in its* $(l+1)$*st column, are preset, and the rest of its entries are independent random bits with probability* $\frac{1}{2}$ *of being zero. What is the probability of* $\mathbf{S}$ *having rank* k?

Let H denote the $k \times l$ matrix consisting of the first l (preset) columns of such a random matrix $\mathbf{S}$. Denote by $\mathbf{T}$ the matrix consisting of the first $l+1$ columns of $\mathbf{S}$ and by $\mathbf{W}$ the matrix consisting of the last $m - l - 1$ columns of $\mathbf{S}$. Also let the random variable ρ denote the rank of $\mathbf{T}$. Clearly, ρ may take only two values, namely, rank (H) or rank $(H) + 1$. We now show that

$$\text{Prob } \{ \text{ rank } (\mathbf{S}) < k|\rho = r\} = 1 - \prod_{i=0}^{k-r-1} (1 - 2^{i+l+1-m}) < 2^{k+l+1-m-r}. \tag{18}$$

Indeed, without loss of generality assume that the first r rows of $\mathbf{T}$ are linearly independent. We assume that the entries of $\mathbf{W}$ are chosen randomly row by row. Having selected the first r rows of $\mathbf{W}$, we thus obtain the first r rows in $\mathbf{S}$ which, in turn, are linearly independent. Next we select the $(r+1)$st row in $\mathbf{W}$. Clearly, there are 2^{m-l-1} choices for such a row, out of which one row will result in an $(r+1)$st row in $\mathbf{S}$ which is spanned by the first r rows in $\mathbf{S}$. Hence, given that the first r rows in $\mathbf{W}$ have been set, the probability that the first $r+1$ rows in $\mathbf{S}$ will be linearly independent is $1 - 2^{l+1-m}$. Conditioning upon the linear independence of the first $r+1$ rows in $\mathbf{S}$, we now select the $(r+2)$nd row in $\mathbf{W}$. In this case there are two choices

of this row that yield an $(r+2)$nd row in $\mathbf{S}$ which is spanned by the first $r+1$ rows in $\mathbf{S}$. Hence, the probability of the first $r+2$ rows in $\mathbf{S}$ to be linearly independent (given the linear independence of the first $r+1$ rows) is $1-2^{l+2-m}$. In general, assuming linear independence of the first $r+i$ rows in $\mathbf{S}$, there are 2^i choices for the $(r+i+1)$st row of $\mathbf{W}$ that yield a row in $\mathbf{S}$ belonging to the linear span of the first $r+i$ rows in $\mathbf{S}$. The conditional probability for the first $r+i+1$ rows in $\mathbf{S}$ to be linearly independent thus becomes $1-2^{i+l+1-m}$. Equation (18) is obtained by reiterating the process for all rows of $\mathbf{W}$.

To complete the computation of the probability of $\mathbf{S}$ having rank k, we need to calculate the probability of ρ being $r = \text{rank}\ (H)$. Let H_t denote the first t rows of H with $r_t \triangleq \text{rank}\ (H_t)$ and let $\mathbf{c}$ denote the first t (preset) entries of the $(l+1)$st column of $\mathbf{S}$ (or of $\mathbf{T}$). We now show that

$$\text{(19)} \qquad \text{Prob}\ \{\rho = r = \text{rank}\ (H)\} = \begin{cases} 2^{r-r_t-k+t} & \text{if rank}\ ([H_t; \mathbf{c}]) = r_t, \\ 0 & \text{if rank}\ ([H_t; \mathbf{c}]) = r_t + 1. \end{cases}$$

We first perform elementary operations on the columns of H so that (i) the first r_t columns in H_t are linearly independent whereas the remaining $l-r_t$ columns in H_t are zero, and (ii) the first r columns in H are linearly independent whereas the remaining $l-r$ columns in H are zero. Now if $\mathbf{c}$ is not in the linear span of the columns of H_t, then $\rho = \text{rank}\ (\mathbf{T}) = \text{rank}\ (H) + 1$. Otherwise, there are 2^{r-r_t} ways to select the last $k-t$ entries of the $(l+1)$st column of $\mathbf{T}$ to have that column spanned by the columns of H: Each such choice corresponds to one linear combination of the last $r-r_t$ nonzero columns of H. Therefore, conditioning upon $\text{rank}\ ([H_t; \mathbf{c}]) = r_t$, the probability of having $\text{rank}\ (T) = \text{rank}\ (H)$ equals 2^{r-r_t-k+t}.

Equations (18) and (19) can be now applied to $\mathbf{S}_u$ to compute the right-hand side of (17), where $\mathcal{A}$ stands for the event of having $n-1$ columns in $\mathbf{U}_n$ set to U_{n-1}, and t bits of the currently added nth column set to $b_1, b_2, \ldots, b_t$.

4.4. Proof using the Lovász local lemma. In this section we present an alternative proof for the existence of a memory allocation $\mathbf{x}$ satisfying (3) and of $k \times x_u$ binary matrices B_u for the encoding mappings $\mathcal{E}_u : \mathbf{w} \mapsto \mathbf{w}B_u$, $u \in V$. The techniques used will turn out to be useful in §4.5. To this end, we make use of the following lemma.

LEMMA 5 (the Lovász local lemma [5], [19]). *Let $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$ be events in an arbitrary probability space. Suppose that each event $\mathcal{A}_i$ is mutually independent of a set of all, but at most δ, events $\mathcal{A}_j$ and that* Prob $\{\mathcal{A}_i\} \le p$ *for all* $1 \le i \le n$. *If* $ep\delta < 1$, *then* Prob $\{\bigwedge_{i=1}^{n} \overline{\mathcal{A}_i}\} > 0$.

In most applications of the lemma (as well as in its use in the sequel), the $\mathcal{A}_i$'s stand for "bad" events; hence, if the probability of each bad event is at most p, and if the bad events are not terribly dependent of one another (in the sense stated in the lemma), there is a strictly positive probability that none of the bad events will occur. However, this probability might be exponentially small. Recently, Beck [2] has proposed a constructive technique that can be used in most applications of the lemma for finding an element of $\bigwedge_{i=1}^{n} \overline{\mathcal{A}_i}$ (see also [1]). We shall be mainly concentrating on an existence proof, as the construction will then follow by a technique similar to the one in [2].

We start by using the local lemma to present an alternative proof of Theorem 3. Given a network graph $G = (V, E_G)$ and an integer m, we construct a directed graph $H = (V, E_H)$ which satisfies the following four properties:

(i) There is an edge $u \to v$ in H whenever u is adjacent to v in G;
(ii) there are no parallel edges in H;
(iii) each node in H has the same in-degree $\Delta_H = O(\Delta_G)$;
(iv) each node in H has an out-degree which is bounded from above by $\Lambda_H = O(\Delta_G)$.

LEMMA 6. *A directed graph H satisfying* (i)–(iv) *always exists.*

Proof. When $\Delta_G > \frac{1}{2}|V|$ we take H as the complete graph (i.e., the adjacency matrix A_H is the all-one matrix and $\Delta_H = \Lambda_H = |V| < 2\Delta_G$). Otherwise, we construct H out of G as follows: Make every self loop in G a directed self loop in H, and change all other edges in G into two antiparallel edges in H. Finally, adjoin extra edges (not parallel to existing ones) to have in-degree $\Delta_H = \Delta_G$ and out-degree $\leq \Lambda_H = 2\Delta_G$ at each node in H. To realize this last step, we scan the nodes of H and add incoming edges to nodes whose in-degree is less than Δ_G—one node at a time. Let u be such a node and let $\overline{\Gamma}(u)$ be the set of nodes in H with no outgoing edges that terminate at u. We show that at least one of the nodes in $\overline{\Gamma}(u)$ has an out-degree less than $2\Delta_G$, thus allowing us to adjoin a new incoming edge to u from that node. The proof then continues inductively. Now, since the in-degree of each node in H at each stage is at most Δ_G, the total number of edges outgoing from nodes in $\overline{\Gamma}(u)$ is bounded from above by $\Delta_G \cdot (|V| - 1)$. On the other hand, $\overline{\Gamma}(u)$ contains at least $|V| - \Delta_G + 1$ nodes. Hence, there exists at least one node in $\overline{\Gamma}(u)$ whose out-degree is at most $(\Delta_G \cdot (|V| - 1))/(|V| - \Delta_G + 1)$; this number, in turn, is less than $2\Delta_G$ whenever $\Delta_G \leq \frac{1}{2}|V|$. □

Proof of Theorem 3 using the local lemma. Let $\mathbf{z}$ be a solution to the linear programming problem LP(G) of (2). By property (i), $\mathbf{z}$ satisfies the inequality $A_H\mathbf{z} \geq \mathbf{1}$. Redefine β_G to be $8e\Delta_G^2$ (and ℓ accordingly to be $m + c_1 \cdot L\{\beta_G, m\}$), and let $\mathbf{X}$ be obtained by (5)–(7). By Proposition 1 we have

$$\text{Prob}\,\{(A_G\mathbf{X})_u < m\} \leq \frac{1}{\beta_G}, \tag{20}$$

and by property (ii) and Proposition 2 we have

$$\text{Prob}\,\{(A_H\mathbf{X})_u > \ell \cdot (A_H\mathbf{z})_u + c_2 \cdot L\{\beta_G, \ell \cdot (A_H\mathbf{z})_u\}\} \leq \frac{1}{\beta_G} \tag{21}$$

for each node $u \in V$.

For every $u \in V$ define the event $\mathcal{A}_u$ as

$$\mathcal{A}_u \triangleq \left\{ \begin{array}{c} (A_G\mathbf{X})_u < m \\ \text{or} \\ (A_H\mathbf{X})_u > \ell \cdot (A_H\mathbf{z})_u + c_2 \cdot L\{\beta_G, \ell \cdot (A_H\mathbf{z})_u\} \end{array} \right\}. \tag{22}$$

By (20) and (21) it follows that Prob $\{\mathcal{A}_u\} \leq 2/\beta_G < 1/(4e\Delta_G^2)$. For every node u in H, denote by $\Gamma_{\text{out}}(u)$ the set of terminal nodes of the edges outgoing from u in H. Then, for every node u, the event $\mathcal{A}_u$ is mutually independent of all events $\mathcal{A}_v$ such that $\Gamma_{\text{out}}(u) \bigcap \Gamma_{\text{out}}(v) = \emptyset$. Hence, by properties (iii) and (iv), each $\mathcal{A}_u$ depends on at most $\Lambda_H(\Delta_H - 1) + 1 \leq 4\Delta_G^2$ events A_v and, therefore, by Lemma 5 there exists a nonnegative integer vector $\mathbf{x}$ satisfying both

$$(A_G\mathbf{x})_u \geq m \tag{23}$$

and

$$(A_G\mathbf{x})_u \leq \ell \cdot (A_H\mathbf{z})_u + c_2 \cdot L\{\beta_G, \ell \cdot (A_h\mathbf{z})_u\} \tag{24}$$

for all $u \in V$.

We now show that $\|\mathbf{x}\|$ satisfies the inequality

$$\frac{\|\mathbf{x}\|}{\|\ell \cdot \mathbf{z}\|} \leq 1 + c_2\left(\frac{\log_e \beta_G}{\ell} + \sqrt{\frac{\log_e \beta_G}{\ell}}\right). \tag{25}$$

By (24) and the fact that each node in H has in-degree Δ_H we have

$$\begin{aligned}\|\mathbf{x}\| &= \frac{1}{\Delta_H}\sum_{u\in V}(A_H\mathbf{x})_u \\ &\le \frac{\ell}{\Delta_H}\sum_{u\in V}(A_H\mathbf{z})_u + \frac{c_2}{\Delta_H}\sum_{u\in V} L\{\beta_G, \ell\cdot(A_H\mathbf{z})_u\} \\ &\le \ell\cdot\|\mathbf{z}\| + c_2\left(\frac{|V|}{\Delta_H}\log_e\beta_G + \frac{1}{\Delta_H}\sum_{u\in V}\sqrt{\ell\cdot(A_h\mathbf{z})_u\cdot\log_e\beta_G}\right).\end{aligned}$$

Now, by the Cauchy–Schwarz inequality,

$$\sum_{u\in V}\sqrt{(A_H\mathbf{z})_u} \le \sqrt{|V|}\cdot\sqrt{\sum_{u\in V}(A_H\mathbf{z})_u} = \sqrt{|V|\cdot\Delta_H\cdot\|\mathbf{z}\|}$$

and therefore

$$\|\mathbf{x}\| \le \ell\cdot\|\mathbf{z}\| + c_2\left(\frac{|V|}{\Delta_H}\log_e\beta_G + \sqrt{\frac{|V|}{\Delta_H}}\cdot\sqrt{\|\mathbf{z}\|\cdot\ell\cdot\log_e\beta_G}\right).$$

Inequality (25) is now obtained by bounding $|V|/\Delta_H$ from above by $\|\mathbf{z}\|$. Finally, Theorem 3 is a consequence of both (23) and (25). □

We now turn to defining the encoding and decoding mappings for a given instange (G, k). To this end, we shall make use of the following lemma.

LEMMA 7. *Let $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_t$ be subsets of $\langle n\rangle \stackrel{\Delta}{=} \{1, 2, \ldots, n\}$, each $\mathcal{S}_i$ of size $\ge s$, and no subset intersects more than δ subsets. Let q be a power of a prime and let k be a nonnegative integer satisfying*

$$e\cdot\delta\cdot q^{-s-1} < q^{-k}.$$

Then there exists an (n, q^k) linear code over $\Phi = GF(q)$, which is separable with respect to each $\mathcal{S}_i$.

Proof. We construct inductively $l \times n$ matrices B_l, $1 \le l \le k$, each generating a linear code which is separable with respect to every $\mathcal{S}_i$; that is, each $(B_l)_{\mathcal{S}_i}$ has rank l. Start with an all-one $1 \times n$ matrix B_1. As the induction step, assume that a matrix B_{l-1}, with the above property, has already been constructed for some $l \le k$. We are now to append at lth row to B_{l-1}.

Given such a matrix B_{l-1}, a row vector in Φ^n is "good" with respect to $\mathcal{S}_i$ if, when appended to B_{l-1}, it yields a matrix B_l such that $(B_l)_{\mathcal{S}_i}$ has rank l; otherwise, a row vector is "bad" with respect to that $\mathcal{S}_i$. Now, for each i, the row span of $(B_{l-1})_{\mathcal{S}_i}$ consists of q^{l-1} vectors in $\Phi^{|\mathcal{S}_i|}$; this means that the probability of a randomly selected row, being bad with respect to $\mathcal{S}_i$, is $q^{-|\mathcal{S}_i|+l-1} \le q^{-s-1+k} < 1/(e\cdot\delta)$. Similarly, if $\mathcal{S}_i \bigcap \mathcal{S}_j = \emptyset$, then the probability of a randomly selected row, being bad with respect to both $\mathcal{S}_i$ and $\mathcal{S}_j$, is $q^{-|\mathcal{S}_i|-|\mathcal{S}_j|+2(l-1)}$. Therefore, when $\mathcal{S}_i \bigcap \mathcal{S}_j = \emptyset$, the events "the row vector is bad with respect to $\mathcal{S}_i$" and "the row vector is bad with respect to $\mathcal{S}_j$" are independent; thus, by Lemma 5 we are guaranteed to have a row vector in Φ^n which is good with respect to every $\mathcal{S}_i$. This vector can now be appended to B_{l-1} to obtain a generator matrix B_l with $(B_l)_{\mathcal{S}_i}$ having rank l for all i. □

Let $\mathbf{x}$ be the integer vector guaranteed by Theorem 3 for $m = k + 2\lceil\log_2\Delta_G\rceil + 1$. Partition the set $\langle\|\mathbf{x}\|\rangle$ into $|V|$ (disjoint) subsets $\mathcal{Q}_u$ with $|\mathcal{Q}_u| = x_u$ and let $\mathcal{S}_u \stackrel{\Delta}{=} \bigcup_{v\in\Gamma(u)}\mathcal{Q}_v$, $u \in V$.

We have $|\mathcal{S}_u| = (A_G\mathbf{x})_u \geq m = k + 2\lceil \log_2 \Delta_G \rceil + 1$ and, therefore, $e \cdot \Delta_G^2 \cdot 2^{-|\mathcal{S}_u|-1} < 2^{-k}$. Furthermore, each $\mathcal{S}_u$ intersects at most $(\Delta_G - 1)^2 + 1$ sets $\mathcal{S}_v$; hence, by Lemma 7 there exists a linear $(\|\mathbf{x}\|, 2^k)$ code over F_2 which is separable with respect to each $\mathcal{S}_u$. For each $u \in V$ let $B_u \triangleq (B)_{\mathcal{Q}_u}$; i.e., B_u is the $k \times x_u$ matrix consisting of all columns of B indexed by $\mathcal{Q}_u$. We now use this to define the encoding and decoding mappings as in §4.2.

4.5. Variations on the memory cost measure. The techniques used in §4.4 can be adapted to obtain file distribution schemes $(G, k) \mapsto \chi(G, k)$ which are close to optimal with respect to other variants of the memory cost measure. For instance, consider the problem where for every instance (G, k), we are looking for a file distribution protocol $\chi(G, k)$ whose memory allocation $\mathbf{x}$ satisfies the following criteria:

(i) The largest component $x_{\max}$ of $\mathbf{x}$ is the smallest possible.

(ii) Among all file distribution protocols that satisfy (i), we take one whose memory size $\|\mathbf{x}\|$ is the smallest.

This variant of our original problem might suit cases where, say, each node in the network graph (as opposed to some "network manager") needs to pay for its own memory. Since the respective decision problem is NP complete, we need to look for approximations to the optimal solution.

Given a network graph $G = (V, E)$ and an integer k, we proceed as follows. Let $\Delta_{\min}$ be $\min_{u \in V} \Delta(u)$. It is clear that $\lceil k/\Delta_{\min} \rceil$ is a lower bound on the largest component of $\mathbf{x}$. Set $\alpha = \lceil k/\Delta_{\min} \rceil / k$ and consider the following linear program:

$$\text{LP}(G; \alpha): \quad \rho_{G;\alpha} = \min \|\mathbf{z}\|, \quad \text{ranging over all rational } \mathbf{z} = [z_u]_{u \in V} \text{ such that } A_G\mathbf{z} \geq \mathbf{1} \quad \text{and} \quad 0 \leq z_u \leq \alpha \quad \text{for every } u \in V. \tag{26}$$

Next, we set $\beta_G = 12e\Delta_G^2$, $m = k + 2\lceil \log_2 \Delta_G \rceil + 1$, and $\ell = m + c_1 \cdot L\{\beta_G, m\}$. Now let $\mathbf{X} = [X_u]_{u \in V}$ be obtained by (5)–(7) and redefine the events $\mathcal{A}_u$ in (22) as

$$\mathcal{A}_u \triangleq \left\{ \begin{array}{c} (A_G\mathbf{X})_u < m \\ \text{or} \\ X_u > \ell \cdot z_u + c_2 \cdot L\{\beta_G, \ell \cdot z_u\} \\ \text{or} \\ (A_H\mathbf{X})_u > \ell \cdot (A_H\mathbf{z})_u + c_2 \cdot L\{\beta_G, \ell \cdot (A_H\mathbf{z})_u\} \end{array} \right\}.$$

By (20) and (21) we have Prob $\{\mathcal{A}_u\} \leq 3/\beta_G < 1/(4e\Delta_G^2)$. Following along the lines of §4.4, the Lovász local lemma now guarantees a file distribution protocol with memory allocation $\mathbf{x}$ whose maximal component $x_{\max}$ and size $\|\mathbf{x}\|$ satisfy both

$$\frac{x_{\max}}{\alpha \cdot k} = 1 + O\left(\max\left\{\frac{\log \Delta_G}{k}; \sqrt{\frac{\log \Delta_G}{k}}\right\}\right)$$

and

$$\frac{\|\mathbf{x}\|}{\rho_{G;\alpha} \cdot k} = 1 + O\left(\max\left\{\frac{\log \Delta_G}{k}; \sqrt{\frac{\log \Delta_G}{k}}\right\}\right).$$

Both $x_{\max}$ and $\|\mathbf{x}\|$ approach their optimal values as k becomes larger than $\log \Delta_G$.

5. The integer programming bound is not tight. In §4 we presented an algorithm for finding a constructive file distribution scheme $(G, k) \mapsto \chi(G, k)$ such that the ratio between the memory size $|\chi(G, k)|$ and $\rho_G \cdot k$ approaches 1 as the ratio $k/\log \Delta_G$ tends to infinity. In this section we present a family of network graphs $\{G_l\}_{l=1}^{\infty}$ for which a file size of $\log \Delta_{G_l}$ is, indeed, a critical point: There exists a sequence of file sizes $k_l \geq \log_2 \Delta_{G_l}$, $l = 1, 2, \ldots$, for which the ratios $M(G_l, k_l)/J(G_l, k_l)$ (and, therefore, $M(G_l, k_l)/(\rho_{G_l} \cdot k_l)$) are bounded away from 1.

For integers m and l, $m \geq l$, define the network graphs $G_{m,l} = (V_{m,l}, E_{m,l})$ as follows: Let U_m be a set of m elements (say, $U_m = \langle m \rangle$) and let $\mathcal{W}_{m,l}$ consist of all subsets of U_m of size l. Set $V_{m,l} = U_m \bigcup \mathcal{W}_{m,l}$ and draw an edge between two nodes $u, v \in V_{m,l}$ in any of the following cases: (i) both u and v are in U_m (i.e., U_m is a clique); (ii) $u \in U_m$, $v \in \mathcal{W}_{m,l}$, and $u \in v$; (iii) $u = v$ (self loops).

First, we verify that $\rho_{G_{m,l}} = m/l$. Let $\mathbf{z} = [z_u]_{u \in V_{m,l}}$ be a nonnegative real vector satisfying $A_{G_{m,l}}\mathbf{z} \geq \mathbf{1}$ and $\|\mathbf{z}\| = \rho_{G_{m,l}}$. Without loss of generality, we can assume that $z_v = 0$ for every $v \in \mathcal{W}_{m,l}$; otherwise, "remove" the quantity z_v from such a node v and add it to the value z_u at some node $u \in \Gamma(v) - \{v\} \subseteq U_m$. This change results in a new nonnegative vector $\tilde{\mathbf{z}}$ with the same norm as $\mathbf{z}$ and which satisfies $A_{G_{m,l}}\tilde{\mathbf{z}} \geq \mathbf{1}$.

Now, rename the nodes of U_m to have $U_m = \langle m \rangle$ and $z_1 \leq z_2 \leq \cdots \leq z_m$. For the node $\langle l \rangle \in \mathcal{W}_{m,l}$ we have

$$\sum_{u=1}^{l} z_u = (A_{G_{m,l}}\mathbf{z})_{\langle l \rangle} \geq 1,$$

and, therefore, $z_u \geq z_l \geq 1/l$ for every node $u \geq l$ in U_m. Hence,

$$\rho_{G_{m,l}} = \|\mathbf{z}\| = \sum_{u=1}^{l} z_u + \sum_{u=l+1}^{m} z_u \geq 1 + \frac{m-1}{l} = \frac{m}{l}.$$

Setting $\mathbf{z} = [z_u]_{u \in V_{m,l}}$ to

$$z_u = \begin{cases} 1/l & \text{if } u \in U_m, \\ 0 & \text{otherwise,} \end{cases}$$

we obtain the equality $\rho_{G_{m,l}} = m/l$. Furthermore,

$$J(G_{m,l}, r \cdot l) = \rho_{G_{m,l}} \cdot r \cdot l = r \cdot m \tag{27}$$

for every positive integer r. A similar analysis for a similar set-covering problem appears also in [7].

In the forthcoming discussion we will be concentrating on two types of network graphs $G_{m,l}$, namely the following:

- $G_l \triangleq G_{2l,l}$, in which case $\rho_{G_l} = 2$ and

$$\log_2 \Delta_{G_l} = \log_2 \left(2l + \binom{2l-1}{l-1}\right) \leq 2l;$$

- $H_l \triangleq G_{2^l,l}$, in which case $\rho_{H_l} = 2^l/l$ and

$$\log_2 \Delta_{H_l} = \log_2 \left(2^l + \binom{2^l-1}{l-1}\right) \leq l^2, \qquad l \geq 2.$$

The proof of the next proposition makes use of the following known lemma.

LEMMA 8 (the sphere-packing or the Hamming bound [12, Chap. 1]). *Let Φ be an alphabet of q elements. There exists an (n, K) code of minimum distance $2t+1$ over Φ only if*

$$K \cdot \sum_{i=0}^{t} \binom{n}{i} (q-1)^i \leq q^n.$$

PROPOSITION 3. *For any fixed positive integer r,*

$$\lim_{l\to\infty} \frac{M(G_l, r\cdot l)}{\rho_{G_l}\cdot r\cdot l} = \lim_{l\to\infty} \frac{M(G_l, r\cdot l)}{J(G_l, r\cdot l)} \geq 1 + \frac{1}{2r}.$$

Proof. Set $k = rl$ for some positive integer r and let $\mathbf{x}$ be the memory allocation of a file distribution protocol χ for (G_l, k) of memory size $|\chi| = \|\mathbf{x}\| = M(G_l, k)$. We assume that $x_v = 0$ for every $v \in \mathcal{W}_{2l,l}$ and that the nodes of $U_{2l} = \langle 2l\rangle$ are renamed to have $x_1 \leq x_2 \leq \cdots \leq x_{2l}$. Letting $h \triangleq x_{l+2}$, we obtain

$$\begin{aligned} M(G_l, k) = \|\mathbf{x}\| &= \sum_{u=1}^{l} x_u + x_{l+1} + \sum_{u=l+2}^{2l} x_u \\ &\geq r\cdot l + r + (l-1)h, \end{aligned} \tag{28}$$

where the inequality follows from $\sum_{u=1}^{l} x_u = (A_{G_l}\mathbf{x})_{\langle l\rangle} \geq k = rl$ which, in turn, implies the inequalities $x_{l+1} \geq x_l \geq r$.

For a file $\mathbf{w} \in F_2^k$, let $\mathbf{c}_\mathbf{w}$ denote the encoded memory contents $[\mathcal{E}_u(\mathbf{w})]_{u=1}^{l+2}$ as determined by the file distribution protocol χ. We now regard the set

$$C \triangleq \{\mathbf{c}_\mathbf{w} \mid \mathbf{w} \in F_2^k\}$$

as an $(l+2, 2^k)$ code over an alphabet of $q \triangleq 2^h$ elements. The code C must be separable with respect to any subset of $\langle l+2\rangle$ of size l, or else there would be nodes in $\mathcal{W}_{2l,l}$ that could not reconstruct the file $\mathbf{w}$. Hence, by Lemma 1, the minimum distance of C is at least 3, which readily implies by Lemma 8 the inequality

$$2^k \cdot (1 + (l+2)(q-1)) \leq q^{l+2}.$$

Substituting $k = rl$ and $q = 2^h$, and noting that $2^h - 1 \geq 2^{h-1}$, we obtain,

$$(l+2)\cdot 2^{rl+h-1} \leq 2^{(l+2)h},$$

or

$$h \geq \left\lceil \frac{\log_2(l+2) + rl - 1}{l+1} \right\rceil = r + \left\lceil \frac{\log_2(l+2) - r - 1}{l+1} \right\rceil.$$

Hence, for fixed r and for sufficiently large l we must have $h \geq r+1$. Combining this lower bound on h with (28) yields the inequality

$$\lim_{l\to\infty} \frac{M(G_l, r\cdot l)}{J(G_l, r\cdot l)} \geq \lim_{l\to\infty} \frac{r(l+1) + (r+1)(l-1)}{2rl} = 1 + \frac{1}{2r}$$

which, with (27), concludes the proof. □

COROLLARY 2. *For* $k_l \triangleq 2l \geq \log_2 \Delta_{G_l}$,

$$\lim_{l\to\infty} \frac{M(G_l, k_l)}{\rho_{G_l} \cdot k_l} = \lim_{l\to\infty} \frac{M(G_l, k_l)}{J(G_l, k_l)} \geq \frac{5}{4}.$$

Corollary 2 exhibits the fact that a file size of $\log \Delta_{G_l}$ is a critical point in the following strong sense: For $k_l = 2l \geq \log_2 \Delta_{G_l}$, the size of any memory allocation for (G_l, k_l) must be bounded away from $\rho_{G_l} \cdot k_l$, not because of a gap between $J(G_l, k_l)$ and $\rho_{G_l} \cdot k_l$, but rather because of a gap between $M(G_l, k_l)$ and $J(G_l, k_l)$.

We point out that, as a counterpart of Proposition 3, we also have

$$\lim_{l\to\infty} \frac{M(G_l, r \cdot l)}{J(G_l, r \cdot l)} \leq 1 + \frac{2}{r},$$

the proof of which is based on the following result.

LEMMA 9 (the Gilbert–Varshamov bound [3, pp. 321–322]). *Let* Φ *be an alphabet of* q *elements and let* n, K, *and* d *be positive integers satisfying*

$$(K-1) \cdot \sum_{i=0}^{d-1} \binom{n}{i} (q-1)^i < q^n.$$

Then there exists an (n, K) *code of minimum distance* d *over* Φ.

Set $n = 2l$, $K = 2^{rl}$, $d = l + 1$, $h + r + 2$, and $q = 2^h$; these values satisfy the equality $K \cdot 2^n \cdot q^{d-1} = q^n$ and, therefore, by Lemmas 1 and 9 there exists a $(2l, 2^{rl})$ code C over F_2^h which is separable with respect to any subset of $\langle 2l \rangle$ of size l. Assign the coordinates (over F_2^h) of C to the nodes $u \in U_{2l}$ of G_l and map the files $\mathbf{w} \in F_2^{rl}$ into distinct codewords of C. This protocol allows every node in G_l to reconstruct any such file $\mathbf{w}$, requiring a total memory size of $2(r+2)l$ (compared to $J(G_l, r \cdot l) = 2rl$).

Remark 5. It can be readily verified that $J(G_{m,l}, k) \geq m - l + k$ for every m, l, and k, and, in particular, $J(G_l, 1) \geq l + 1 \geq \frac{1}{4}\rho_{G_l} \log_2 \Delta_{G_l}$. Hence, any file distribution protocol for (G_l, k) based on segmentation will be at least $\frac{1}{4}\log_2 \Delta_{G_l}$ times larger than the linear programming bound $\rho_{G_l} \cdot k$, *even when* k *tends to infinity* (see Example 1 and Remark 3).

For file sizes k which are smaller than $\log \Delta_G$, one can find examples where the ratio between $M(G, k)$ and $J(G, k)$ is even larger than stated in Proposition 3. We demonstrate this for the network graphs $H_l = G_{2^l,l}$ in the next proposition, making use of the following lemma.

LEMMA 10 (the Plotkin bound [3, p. 315]). *Let* C *be an* (n, K) *code of minimum distance* d *over an alphabet of* q *elements. Then*

$$\frac{1}{q} \leq 1 - \frac{d}{n}\left(1 - \frac{1}{K}\right).$$

PROPOSITION 4.

$$\frac{M(H_l, k)}{J(H_l, k)} \sim \begin{cases} 1 & \text{if } l|k \text{ and } k \geq l^2, \\ l^2/k & \text{if } l|k \text{ and } l \leq k < l^2, \\ k & \text{if } k < l, \end{cases}$$

where $f_l(k) \sim g_l(k)$ *stands for* $\lim_{l\to\infty} f_l(k)/g_l(k) = 1$ *uniformly on* k.

In particular, when $k = l$, the ratio $M(H_l, k)/J(H_l, k)$ is approximately l which, in turn, is at least $\sqrt{\log_2 \Delta_{H_l}}$.

Proof. We distinguish between the three ranges of k stated in the proposition.

Case 1. $k = rl$ for some integer $r \geq l$. By (27) we have $J(H_l, k) = \rho_{H_l} \cdot k = r \cdot 2^l$. In fact, we also have $M(H_l, k) = J(H_l, k)$: Since $r \geq l$, we can construct a $(2^l, 2^{rl})$ generalized Reed–Solomon code C_{RS} over $GF(2^r)$, which is separable with respect to any subset of $\langle 2^l \rangle$ of size l [12, Chaps. 10–11] (compare with §3.3). Assign the coordinates, over $GF(2^r)$, of C_{RS} to the nodes $u \in U_{2^l}$ of H_l and map the files $\mathbf{w} \in F_2^k$ into distinct codewords of C_{RS}. By the separability of C_{RS} every node in H_l can readily reconstruct any file $\mathbf{w} \in F_2^k$.

Case 2. $k = rl$ for some strictly positive integer $r < l$. Let $\mathbf{x}$ be the memory allocation of a file distribution protocol for (H_l, k) of memory size $\|\mathbf{x}\| = M(H_l, k)$. Again, we assume that $x_v = 0$ for $v \in \mathcal{W}_{2^l, l}$ and that the nodes in U_{2^l} are renamed to have $x_1 \leq x_2 \leq \cdots \leq x_{2^l}$. Defining $n \triangleq \lceil 2^l / l \rceil$ and $h \triangleq x_{n+1}$, we obtain

$$M(H_l, k) = \|\mathbf{x}\| \geq \sum_{u=n+1}^{2^l} x_u \geq (2^l - n)h > h \cdot 2^l \cdot \left(1 - \frac{1}{l} - \frac{1}{2^l}\right) \tag{29}$$

(compare with (28)).

To bound h from below, we regard the set

$$C \triangleq \{[\mathcal{E}_u(\mathbf{w})]_{u=1}^n \mid \mathbf{w} \in F_2^k\}$$

as an $(n, 2^k)$ code over an alphabet of $q \triangleq 2^h$ elements. Since C is separable with respect to any subset of $\langle n \rangle$ of size l, its minimum distance must be, by Lemma 1, at least $n - l + 1$. This, in turn, implies by Lemma 10 the inequality

$$\frac{1}{2^h} \leq 1 - \left(1 - \frac{l-1}{n}\right)\left(1 - \frac{1}{2^k}\right) \leq \frac{l-1}{n} + \frac{1}{2^k}. \tag{30}$$

Since $2^k \geq 2^l \geq n$, we thus have

$$\frac{1}{2^h} \leq \frac{l}{n} \leq \frac{l^2}{2^l}$$

or

$$h \geq l - 2\log_2 l.$$

Combining the last inequality with (29) yields

$$M(H_l, k) > 2^l \cdot (l - 2\log_2 l)\left(1 - \frac{1}{l} - \frac{1}{2^l}\right) = l \cdot 2^l \cdot (1 - o(1)), \tag{31}$$

where $o(1)$ stands for an expression, independent of k, which tends to zero as l goes to infinity. Recalling that $J(H_l, k) = r \cdot 2^l$, we thus obtain

$$\frac{M(H_l, k)}{J(H_l, k)} \geq \frac{l}{r} \cdot (1 - o(1)) = \frac{l^2}{k} \cdot (1 - o(1)). \tag{32}$$

The bounds (31) and (32) are definitive up to a multiplying factor of $1 - o(1)$: An upper bound $M(H_l, k) \leq l \cdot 2^l$ is obtained by assigning the coordinates of a $(2^l, 2^{rl})$ generalized Reed–Solomon code over $GF(2^l)$ to the nodes $u \in U_{2^l}$ of H_l; such a code is separable with respect to any subset of $\langle 2^l \rangle$ of size r and, therefore, with respect to any subset of size l.

Case 3. $k < l$. Let n and h be defined as in Case 2. Noting that (29) and (30) still apply, we have

$$\frac{1}{2^h} \leq \frac{l-1}{n} + \frac{1}{2^k} < 2\max\left\{\frac{l^2}{2^l}; \frac{1}{2^k}\right\},$$

i.e.,

$$h \geq \min\{l - \lceil 2\log_2 l\rceil; k\} \geq \left(1 - \frac{\lceil 2\log_2 l\rceil}{l}\right)\cdot k.$$

Combining the last inequality with (29) yields

$$M(H_l, k) \geq k \cdot 2^l \cdot (1 - o(1))\,. \tag{33}$$

Turning to $J(H_l, k)$, for $k < l$ we have $J(H_l, k) \geq 2^l - l + k$ (and by Remark 5 we have, in fact, equality): it is easy to verify that the integer vector $\mathbf{y} = [y_u]_{u\in V_{2^l,l}}$ which is defined by

$$y_u = \begin{cases} 1 & \text{if } u \in U_{2^l} \quad \text{and} \quad u \geq l - k + 1, \\ 0 & \text{otherwise,} \end{cases}$$

satisfies the inequality $A_{H_l}\mathbf{y} \geq k \cdot \mathbf{1}$. Hence, by (33) we obtain

$$\frac{M(H_l, k)}{J(H_l, k)} \geq k \cdot (1 - o(1))\,. \tag{34}$$

Again, the bounds (33) and (34) are definitive as we can simply replicate the file $\mathbf{w}$ into each node $u \in U_{2^l}$ of H_l, requiring a memory size of $k \cdot 2^l$. □

Appendix. The proofs of Propositions 1 and 2 make use of the following lemma.

LEMMA 11. *Let* $\mathbf{a} = [a_u]_{u\in V}$ *and* $\mathbf{p} = [p_u]_{u\in V}$ *be real vectors in* $[0, 1]^{|V|}$ *and let* $\mathbf{Y} = [Y_u]_{u\in V}$ *be a vector of independent random variables over* $\{0, 1\}$ *with* Prob $\{Y_u = 1\} = p_u$. *Then we have the following results*:

(a) *for every* $\delta \in [0, 1)$ *and* $\tau \leq \mathbf{a}\cdot\mathbf{p}$,

$$\text{Prob}\,\{\mathbf{a}\cdot\mathbf{Y} \leq (1-\delta)\tau\} \leq \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{\tau};$$

(b) *for every* $\delta \geq 0$ *and* $\tau \geq \mathbf{a}\cdot\mathbf{p}$,

$$\text{Prob}\,\{\mathbf{a}\cdot\mathbf{Y} \geq (1+\delta)\tau\} \leq \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\tau}.$$

Proof. Lemma 11 is proved in [15] and [16]. Part (b) of the lemma appears as is in [15] and for the sake of completeness we include the proof of part (a) here.

For a real random variable Z and constants $\gamma \geq 0$ and b, we have

$$\text{Prob}\,\{Z \leq b\} \leq E(e^{\gamma(b-Z)}),$$

an inequality known as the Chernoff bound. Letting $Z = \mathbf{a}\cdot\mathbf{Y}$ and $b = (1-\delta)\tau$, we obtain, for every $\gamma \geq 0$,

$$\begin{aligned}
&\text{Prob}\,\{\mathbf{a}\cdot\mathbf{Y} \leq (1-\delta)\tau\} \\
&\leq E\left(e^{\gamma((1-\delta)r - \mathbf{a}\cdot\mathbf{Y})}\right) = e^{\gamma(1-\delta)\tau} E\left(\prod_{u\in V} e^{-\gamma a_u Y_u}\right) \\
&= e^{\gamma(1-\delta)\tau} \prod_{u\in V} E\left(e^{-\gamma a_u Y_u}\right) = e^{\gamma(1-\delta)\tau} \prod_{u\in V} \left(1 - p_u + p_u e^{-\gamma a_u}\right).
\end{aligned}$$

Substituting $t = e^{-\gamma}$ yields, for every $t \in (0, 1]$,

$$\text{Prob}\{\mathbf{a} \cdot \mathbf{Y} \leq (1-\delta)\tau\} \leq \alpha(t), \tag{35}$$

where

$$\alpha(t) \triangleq t^{-(1-\delta)\tau} \prod_{u \in V}(1 - p_u(1 - t^{a_u})) \leq t^{-(1-\delta)\tau} \prod_{u \in V} \exp\{-p_u(1 - t^{a_u})\}$$

$$= t^{-(1-\delta)\tau} \exp\left\{-\sum_{u \in V} p_u(1 - t^{a_u})\right\}.$$

Now, for $a_u \in [0, 1]$ and $t \in (0, 1]$ we have $1 - t^{a_u} \geq a_u(1 - t) \geq 0$. Therefore,

$$\alpha(t) \leq t^{-(1-\delta)\tau} \exp\left\{-\sum_{u \in V} a_u p_u(1 - t)\right\} \leq (t^{-(1-\delta)} e^{t-1})^{\tau}$$

which, for $t = 1 - \delta$ becomes

$$\alpha(1 - \delta) \leq \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{\tau}.$$

Part (a) is now obtained by substituting $t = 1 - \delta$ in (35). □

Proof of Proposition 1. Let r denote the difference $\ell - m$ and let

$$\tau \triangleq \ell - \ell \cdot \mathbf{a} \cdot \mathbf{z} + \mathbf{a} \cdot \mathbf{p}. \tag{36}$$

Note that $\mathbf{a} \cdot \mathbf{z} \geq 1$ implies $\tau \geq \mathbf{a} \cdot \mathbf{p}$ and that $\mathbf{a} \cdot \mathbf{p} \leq \ell \cdot \mathbf{a} \cdot \mathbf{z}$ implies $\tau \leq \ell$. Also, let $\mathbf{Y}$ be the random variable as in (7). Then,

$$\begin{aligned}\text{Prob}\{\mathbf{a} \cdot \mathbf{X} < m\} &= \text{Prob}\{\mathbf{a} \cdot \mathbf{Y} + \mathbf{a} \cdot \mathbf{s} < \ell - r\} \\ &= \text{Prob}\{\mathbf{a} \cdot \mathbf{Y} < \ell - \ell \cdot \mathbf{a} \cdot \mathbf{z} + \mathbf{a} \cdot \mathbf{p} - r\} \\ &= \text{Prob}\{\mathbf{a} \cdot \mathbf{Y} < \tau - r\},\end{aligned}$$

which readily proves the proposition for $r \geq \tau$. Hence, we assume from now on that $0 \leq r < \tau$.

Apply Lemma 11(a) with τ as in (36) and with $\delta = r/\tau$ (note that, indeed, $\tau \leq \mathbf{a} \cdot \mathbf{p}$). Defining $\sigma \triangleq \tau - r (> 0)$, we thus obtain,

$$\text{Prob}\{\mathbf{a} \cdot \mathbf{Y} < \tau - r\} \leq \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\tau} = \left(1 + \frac{r}{\sigma}\right)^{\sigma} e^{-r}.$$

Therefore, to have $\text{Prob}\{\mathbf{a} \cdot \mathbf{X} < m\} \leq \frac{1}{\beta}$ it suffices to require that

$$r - \sigma \cdot \log_e\left(1 + \frac{r}{\sigma}\right) \geq \log_e \beta. \tag{37}$$

Case 1. $r \leq 2\sigma$. It is easy to verify that $\log_e(1 + t) \leq t - (t^2/6)$ whenever $0 \leq t \leq 2$. Hence, (37) is implied by

$$r - \sigma \cdot \left(\frac{r}{\sigma} - \frac{r^2}{6\sigma^2}\right) \geq \log_e \beta$$

which, in turn, is satisfied if $r \geq \sqrt{6\sigma \log_e \beta}$. Recalling that $\sigma = \tau - r \leq \ell - r = m$, inequality (37) is thus implied by

$$r \geq \sqrt{6m \log_e \beta}. \tag{38}$$

Case 2. $r > 2\sigma$. In this range,

$$\sigma \cdot \log_e \left(1 + \frac{r}{\sigma}\right) = r \cdot \log_e \left(\left(1 + \frac{r}{\sigma}\right)^{(\sigma/r)}\right) \leq r \cdot \log_e \sqrt{3}.$$

Hence, inequality (37) is satisfied if

$$r \geq \frac{\log_e \beta}{1 - \log_e \sqrt{3}}. \tag{39}$$

The existence of the constant c_1 is now implied by (38) and (39) (setting $c_1 = 2.5$ will do). □

Proof of Proposition 2. Let $\mathbf{Y}$ be the random variable as in (7) and let r be a positive number. Then

$$\text{Prob}\{\mathbf{a} \cdot \mathbf{X} > E(\mathbf{a} \cdot \mathbf{X}) + r\} = \text{Prob}\{\mathbf{a} \cdot \mathbf{Y} > \mathbf{a} \cdot \mathbf{p} + r\}.$$

Now the proposition holds trivially when $\mathbf{a} \cdot \mathbf{p} = 0$, since, in this case, Prob $\{\mathbf{a} \cdot \mathbf{Y} = 0\} = 1$. Therefore, we assume from now on that $\mathbf{a} \cdot \mathbf{p} > 0$.

Apply Lemma 11(b) with $\tau = \mathbf{a} \cdot \mathbf{p}$ and $\delta = r/(\mathbf{a} \cdot \mathbf{p})$; we obtain

$$\text{Prob}\{\mathbf{a} \cdot \mathbf{Y} > \mathbf{a} \cdot \mathbf{p} + r\} \leq ((1+\delta)^{-(1+\delta)} e^{\delta})^{\tau}.$$

Therefore, to have Prob $\{\mathbf{a} \cdot \mathbf{X} > E(\mathbf{a} \cdot \mathbf{X}) + r\} \leq \frac{1}{\beta}$ it suffices to require that

$$\tau \cdot ((1+\delta)\log_e(1+\delta) - \delta) \geq \log_e \beta. \tag{40}$$

Case 1. $r/(\mathbf{a} \cdot \mathbf{p}) = \delta \leq \frac{3}{2}$. Noting that $(1+t)\log_e(1+t) \geq t + (t^2/4)$ for $0 \leq t \leq \frac{3}{2}$, inequality (40) is satisfied whenever

$$\frac{\delta^2 \tau}{4} = \frac{r^2}{4(\mathbf{a} \cdot \mathbf{p})} \geq \log_e \beta$$

which, with $E(\mathbf{a} \cdot \mathbf{X}) \geq \mathbf{a} \cdot \mathbf{p}$, is implied by

$$r \geq 2\sqrt{E(\mathbf{a} \cdot \mathbf{X}) \cdot \log_e \beta}. \tag{41}$$

Case 2. $r/(\mathbf{a} \cdot \mathbf{p}) = \delta > \frac{3}{2}$. Noting that $t \mapsto (1 + t^{-1})\log_e(1+t)$ is monotonously increasing for $t > 0$, we have

$$\begin{aligned} \tau \cdot ((1+\delta)\log_e(1+\delta) - \delta) &= r \cdot ((1+\delta^{-1})\log_e(1+\delta) - 1) \\ &\overset{\delta \geq \frac{3}{2}}{\geq} r \cdot \left(\left(\frac{5}{3}\log_e \frac{5}{2}\right) - 1\right) > \frac{r}{2}; \end{aligned}$$

i.e., inequality (40) is satisfied if

$$r \geq 2\log_e \beta. \tag{42}$$

The existence of the constant c_2 (such as $c_2 = 2$) is now implied by (41) and (42). □

Acknowledgment. We thank Noga Alon and Cynthia Dwork for the many helpful discussions. We also thank the anonymous referee for the useful comments and the suggestion that we consider other variations on the memory cost measure.

REFERENCES

[1] N. ALON, *A parallel algorithmic version of the Local Lemma*, Random Structures Algorithms, 2 (1991), pp. 367–378.

[2] J. BECK, *An algorithmic approach to the Lovász Local Lemma*, I, Random Structures Algorithms, 2 (1991), pp. 343–365.

[3] E. R. BERLEKAMP, *Algebraic Coding Theory*, McGraw-Hill, New York, 1968.

[4] L. W. DOWDY AND D. V. FOSTER, *Comparative models of the file assignment problem*, Comp. Surveys, 14 (1982), pp. 287–313.

[5] P. ERDÖS AND L. LOVÁSZ, *Problems and results on 3-chromatic hypergraphs and some related questions*, in Infinite and Finite Sets, A. Hajual et al. eds., Colloq. Math. Soc. J. Bolyai, vol. 11, North Holland, Amsterdam, 1975, pp. 609–627.

[6] M. GAREY AND D. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, CA, 1979.

[7] D. S. HOCHBAUM, *On the fractional solution to the set covering problem*, SIAM J. Alg. Disc. Meth., 4 (1983), pp. 221–222.

[8] G. KANT AND J. VAN LEEUWEN, *File distribution problem for processor networks*, Proc. Scandinavian Workshop on Algorithmic Theory, 1990, pp. 47–59.

[9] N. KARMARKAR, *A new polynomial-time algorithm for linear programming*, Combinatorica, 4 (1984), pp. 373–395.

[10] E. D. KARNIN, J. W. GREENE, AND M. H. HELLMAN, *On secret sharing systems*, IEEE Trans. Inform. Theory, 29 (1983), pp. 35–41.

[11] L. LOVÁSZ, *On the ratio of optimal integral and fractional covers*, Discrete Math., 13 (1975), pp. 383–390.

[12] F. J. MACWILLIAMS AND N. J. A. SLOANE, *The Theory of Error-Correcting Codes*, North-Holland, Amsterdam, 1977.

[13] S. MAHMOUD AND J. S. RIORDAN, *Optimal allocation of resources in distributed information networks*, ACM Trans. Database Systems, 1 (1976), pp. 66–78.

[14] M. O. RABIN, *Efficient dispersal of information for security, load balancing, and fault tolerance*, J. ACM, 36 (1989), pp. 335–348.

[15] P. RAGHAVAN, *Probabilistic construction of deterministic algorithms: approximating packing integer programs*, J. Comp. Sys. Sciences, 37 (1988), pp. 130–143. (See also *Proc. 27th IEEE Symp. Found. Comp. Science* (1986), pp. 10–18.)

[16] ———, *Lecture notes on randomized algorithms*, Technical Report RC 15340 (#68237), IBM T. J. Watson Research Center, 1990.

[17] P. RAGHAVAN AND C. D. THOMPSON, *Randomized rounding: a technique for provably good algorithms and algorithmic proofs*, Combinatorica, 7 (1987), pp. 365–374.

[18] A. SHAMIR, *How to share a secret*, Comm. ACM, 22 (1979), pp. 612–613.

[19] J. SPENCER, *Ten Lectures on the Probabilistic Method*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1987.

SIAM J. COMPUT.
Vol. 24, No. 1, pp. 184–201, February 1995

012

AN OPTIMAL ALGORITHM FOR COMPUTING VISIBILITY IN THE PLANE*

PAUL J. HEFFERNAN† AND JOSEPH S. B. MITCHELL‡

Abstract. The authors give an algorithm to compute the visibility polygon from a point among a set of h pairwise-disjoint polygonal obstacles with a total of n vertices. The algorithm uses $O(n)$ space and runs in optimal time $\Theta(n + h \log h)$, improving the previous upper bound of $O(n \log n)$. A direct consequence of the algorithm is an $O(n + h \log h)$ time algorithm for computing the convex hull of h disjoint simple polygons.

Key words. visibility, hidden line elimination, lower envelopes, computational geometry

AMS subject classifications. 68U05, 68Q25

1. Introduction.

Let $\mathcal{D}$ be a planar polygonal domain with h holes and n vertices: $\mathcal{D}$ is a connected closed subset of the plane whose boundary consists of a set of n line segments. If $h = 0$, then $\mathcal{D}$ is simply connected and is called a *simple polygon*. If $h > 0$, then $\mathcal{D}$ is multiply connected, and its holes form a set $\mathcal{P} = \{P_1, P_2, \ldots, P_h\}$ of pairwise-disjoint simple polygons in the plane. The *visibility polygon* with respect to a point $s \in \mathcal{D}$ is the locus of all points $q \in \mathcal{D}$ such that $\overline{sq} \subset \mathcal{D}$. The problem of computing the visibility polygon with respect to a given point s is known as the "hidden line removal" problem and is fundamental in computational geometry.

Our Result. We provide an algorithm to compute a visibility polygon in optimal time $\Theta(n + h \log h)$ and space $O(n)$.

Relation to Previous Work. Algorithms to compute the visibility polygon have been known for some time; a clear summary of the many known visibility algorithms is given in Chapter 8 of O'Rourke's book [O'R]. For the case of a simple polygon P, optimal $O(n)$ algorithms have been given by [EA], [Le2], and [JS], who correct a minor error in [EA] and [Le2] while simplifying the algorithm of [Le2]. For the case of a polygon P with holes, straightforward $O(n \log n)$-time algorithms can be based on plane (rotational) sweep about s (as in [Le1] and [SO]) or based on divide-and-conquer (as in [AM]). In fact, by using the linear-time algorithms for simple polygons to compute the visible portion of the boundary of each hole, and then merging these "profiles," one can obtain a simple $O(n \log h)$ algorithm for computing the visibility polygon (see [AM] [AAGHI], and [As]).

There is an $\Omega(n + h \log h)$ lower bound (from sorting) for computing a visibility polygon [O'R], [SO]. Optimal algorithms that achieve this time bound were known for the special case in which the holes P_i are convex [AM], [As] or star-shaped [AM]. The question of whether or not an algorithm exists for the *general* case whose running time is *linear* in n has been a fundamental open problem. No algorithm was previously known with running time $O(n + f(h))$ for *any* function $f(h)$.

We resolve the open question by providing a few different methods, culminating in an optimal algorithm. We outline our approach below. We let $\tau(n)$ denote the time required to triangulate a simple polygon. By Chazelle's recent breakthrough [Ch], we know that

*Received by the editors October 7, 1991; accepted for publication (in revised form) October 27, 1993.

†Department of Mathematical Sciences, Memphis State University, Memphis, Tennessee 38152 (hefferna@next1.msci.memst.edu). This work was conducted while the author was supported by a National Science Foundation graduate fellowship at Cornell University.

‡Applied Mathematics and Statistics, State University of New York, Stony Brook, New York 11794-3600 (jsbm@ams.sunysb.edu). Partially supported by a grant from Hughes Research Laboratories, by a grant from Boeing, by NSF Grants ECSE-8857642 and CCR-9204585, and by Air Force Office of Scientific Research contract AFOSR-91-0328.

$\tau(n) = O(n)$, and that there are several fast deterministic and randomized algorithms giving bounds of $O(n \log\log n)$ [KKT], [TV] or $O(n \log^* n)$ [CTV], [Se].

(1) We give a very simple $O(n + h^2)$ algorithm that does not require triangulation.
(2) The algorithm of (1) can be transformed into a dynamic insertion procedure with time complexity $O(\tau(n) + h^2 \log \bar{n})$. Here, $\bar{n} \leq n$ is the number of sides of the most complex hole.
(3) We give an $O(\tau(n) + h \log^2 h)$ algorithm that is relatively simple but relies on triangulation.
(4) We give an $O(\tau(n) + h \log(\bar{n} + h))$ algorithm for the special case in which all of the holes are *stabbed* by a line.
(5) We give an $O(\tau(n) + h \log h + h \log\log h \log^2 \bar{n})$ algorithm based on applying the result (4) to $O(\log h)$ classes of obstacles, and then merging the resulting set of visibility polygons.
(6) Finally, we show how the algorithm of (5) can be modified to yield an optimal time bound of $\Theta(n + h \log h)$ for the general problem.
(7) A direct consequence of our algorithm is an $O(n + h \log h)$ time algorithm for computing the convex hull of h disjoint simple polygons.

Concurrent with our work, [BG] have given an $O(\tau(n) + h^{1+\epsilon})$ algorithm for "merging" the h holes (thereby permitting linear-time visibility computation). Most recently, [BC] have tightened the bound of [BG] to $O(\tau(n) + h \log^{1+\epsilon} h)$.

The remainder of this paper is organized as follows. In the next section we reduce the problem of computing a visibility polygon in a polygonal domain to an equivalent problem that we find easier to work with: computing the visibility profile of disjoint polygons. Section 3 establishes notation and basic properties. In §4, we give our $O(n + h^2)$ algorithm. In §5 we describe lid queries, a type of planar point location that is used in all of our subsequent algorithms. We give the dynamic insertion algorithm in §6, the $O(n + h \log^2 h)$ algorithm in §7, and the optimal $\Theta(n + h \log h)$ algorithm in §8. Section 9 is the conclusion.

2. Reduction to the visibility profile problem. Computing the visibility polygon is known to be equivalent to an alternate problem, that of computing the *visibility profile* from below of a collection of polygons, the so-called "lower envelope." Formally, the visibility profile of a collection of polygons can be described as a function $y = f(x)$ defined over the domain $\mathbb{R}$, where $f(x)$ is the minimum y coordinate of a point of the polygons with x coordinate x ($f(x) = \infty$ if no such point exists). For ease of exposition, we concentrate on the problem of computing visibility profiles. We justify this choice by giving a reduction of the visibility polygon problem to the visibility profile problem. Since the visibility polygon of the outer boundary of $\mathcal{D}$ can be computed in $O(n)$ time, and it can be merged with that of the holes in $O(n)$ time, we restrict our attention to the holes.

The basic idea of the reduction is to center a polar coordinate system, (θ, r), at s, and map the holes into an orthogonal coordinate system, (x, y), where, for a point p of a polygonal hole, $y(p) = r(p)$ and $x(p) = \theta(p) + 2\pi k$ (for some integer k). A line of sight in the polar system, a ray with terminus s, corresponds to a vertical line directed upward in the orthogonal system. If a hole does not intersect the ray $\theta = 0$, then we let the integer k equal 0 for all points on the hole. The situation becomes more complicated when a hole intersects $\theta = 0$; we will need to cut such a hole into two pieces. If P is such a hole, let t and t' be the points of $P \cap \{(\theta, r) | \theta = 0\}$ with minimum and maximum r coordinates, respectively. For a point $p \in P$, we define a value $x^*(p)$ to be the amount of winding (*not* taken mod 2π) on $P_{CW}(t, p)$, the clockwise subchain from t to p (note that $x^*(p) = \theta(p) + 2\pi k$, for some k). Lemma 2.1 (below) tells us that if we set $x(p) = x^*(p)$ for points $p \in P_{CW}(t, t')$, and $x(p) = x^*(p) + 2\pi$ for $p \in P_{CW}(t', t)$, then the visibility profile of P in the orthogonal system

over the x range $[0, 2\pi]$ corresponds to the visible portion of P around s. We have replaced P with two polygonal chains; these chains can be made into polygons if we "double" each vertex. While we have not mapped segments into straight-line segments, we have preserved the basic properties necessary to compute visibility. The visibility polygon problem in the polar system has now been reduced to computing the visibility profile from below and outputting the portion over the x range $[0, 2\pi]$.

LEMMA 2.1. *On subchain $P_{CW}(t, t')$, only points p with $x^*(p) \in [0, 2\pi)$ can be visible from s in the polar system, and on $P_{CW}(t', t)$, only points p with $x^*(p) \in (-2\pi, 0]$ can be visible.*

Proof. That no point $p \in P_{CW}(t, t')$ with $x^*(p) \geq 2\pi$ can be visible follows from the simplicity of $P_{CW}(t, t')$ and the fact that t is visible. To complete the proof, it suffices to show that no point $p \in P_{CW}(t', t)$ with $x^*(p) \geq 0$ can be visible. Consider that the sets $\{p \in P_{CW}(t, t') | x^*(p) \geq 0\}$ and $\{p \in P_{CW}(t', t) | x^*(p) \geq 0\}$ consist of collections of polygonal chains, with endpoints on the ray $\theta = 0$. Furthermore, any chain from $P_{CW}(t', t)$ must have its endpoints nested inside those of a chain from $P_{CW}(t, t')$, and therefore cannot see s. □

3. Notation and basic properties. Let $x(p)$ and $y(p)$ denote the x and y coordinates of a point p in the plane, and that $\rho_d(p)$ and $\rho_u(p)$ represent the rays with root p in the direction straight down and straight up, respectively. If a value x represents an x coordinate, then let x^- and x^+ denote the x coordinate values infinitesimally left and right of x, respectively.

We address the problem of computing the visibility profile from below of a collection of disjoint polygons, $\mathcal{P} = \{P_1, \ldots, P_h\}$. A polygon P_i contains two vertices, l_i and r_i, of a minimum and maximum x coordinate, respectively. Since we assume that the observer is at $y = -\infty$, the chain obtained by traversing P_i clockwise from l_i to r_i is completely blocked from the view of the observer by the counterclockwise chain of P_i from l_i to r_i. We therefore use only this lower chain of P_i when computing the visibility profile. In fact, in the remainder of this paper, we assume that $P_1, \ldots, P_h$ are polygonal chains joining their left endpoints (l_i) to their right endpoints (r_i).

For any set $S \subseteq \mathcal{P} = \{P_1, \ldots, P_h\}$, we let $VP(S)$ denote the visibility profile of the chains in S, and we let $VP(S; x)$ denote the point of $VP(S)$ with x coordinate x. We slightly abuse notation and write $VP(i, \ldots, j)$ and $VP(i, \ldots, j; x)$ instead of $VP(\{P_i, \ldots, P_j\})$ and $VP(\{P_i, \ldots, P_j\}; x)$. If $x \notin [x(l_{\min}), x(r_{\max})]$, where $x(l_{\min}) = \min_{i \in S} x(l_i)$ and $x(r_{\max}) = \max_{i \in S} x(r_i)$, then $VP(S; x)$ is the point (x, ∞).

We can think of the profile $VP(S)$ as a piecewise-continuous function over $\mathbb{R}$. The points x of discontinuity of $VP(S)$ are of two types:

- x is a *jump* if $VP(S)$ coincides with the same chain P_i at both x^- and x^+;
- x is a *leap* if $VP(S)$ coincides with distinct chains P_i and P_j ($i \neq j$) at x^- and x^+.

A maximal connected subdomain of $[x(l_{\min}), x(r_{\max})]$ that does not contain a leap in its interior is called a *piece* (the corresponding section of $VP(S)$ over this domain is also called a piece). Since a piece of $VP(S)$ corresponds to a section of a specific chain P_i, we say that P_i *appears* in $VP(S)$ with this piece. These definitions are illustrated in Fig. 1.

A special type of piece is a "piece at infinity"; that is, a maximal x interval over which no member of S appears. Pieces at infinity sometimes create special cases for our algorithms, but ones that can be handled easily.

While we have thought of $VP(S)$ as a function in order to define jumps and leaps, our algorithms will store a visibility profile $VP(S)$ as a polygonal chain. A vertical edge of the chain $VP(S)$ corresponds to a jump or leap. We call the vertical edge of a jump of a profile $VP(i)$ a *lid*, since its interior is disjoint from P_i. We will usually represent a lid by $\overline{ab}$, where $y(a) < y(b)$; therefore, if $\overline{ab}$ is a lid of $VP(i)$, then $VP(i; x(a)) = a$.

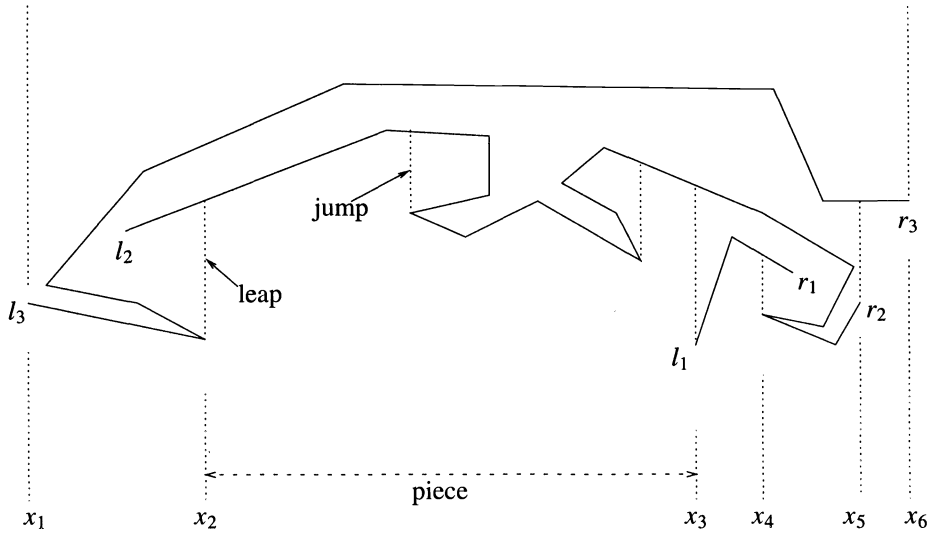


FIG. 1. *Definition of jumps, leaps, and pieces.*

A leap x between chains P_i and P_j can occur in one of two manners. The leap x may be caused by the left or right endpoint of one of the two chains, or it may occur where one profile intersects a lid of the other. These cases are illustrated in Fig. 1: Coordinates x_1 and x_3 correspond to leaps at a left endpoint, x_5 and x_6 correspond to leaps at a right endpoint, and x_2 and x_4 correspond to leaps at lids.

We will often use the expression "p is below q" to indicate that $y(p) < y(q)$. Similar use is made of the terms "above," "left," and "right." We say that profile $VP(S)$ is below profile $VP(S')$ at x coordinate x if $y(VP(S; x^-)) < y(VP(S'; x^-))$ *or* if $y(VP(S; x^+)) < y(VP(S'; x^+))$. It is possible for one but not both of these conditions to hold if one of the profiles has a jump, leap, or endpoint at x.

We will assume without loss of generality that all chains $P_1, \ldots, P_h$ lie completely above the x axis so that the point $p = (x, 0)$ is below $VP(i; x)$ for any x and any profile $VP(i)$.

LEMMA 3.1. *If $x(l_i) < x(l_j)$, then P_j appears at most once in $VP(i, j)$, that is, at most one piece of $VP(i, j)$ is contributed by $VP(j)$.*

Proof. Suppose that $VP(j)$ contributes two pieces to $VP(i, j)$. Let p and q be points of $VP(j)$ on each of the two pieces, with p on the left piece and q on the right piece. Let r be a point of $VP(i)$ that lies on a piece between the two pieces contributed by $VP(j)$, so that $x(p) < x(r) < x(q)$. Refer to Fig. 2.

Now consider the closed Jordan curve given by starting at point $(x(p), 0)$, going up to p, following chain P_j to q, going down to $(x(q), 0)$, and then returning to $(x(p), 0)$ along the x axis. Since point r is on the profile $VP(i, j)$, it must lie in the bounded component defined by this Jordan curve. On the other hand, the leftmost point l_i must lie in the unbounded component defined by the closed curve, since l_i is to the left of l_j. This implies that P_i must cross the Jordan curve, which is a contradiction, since p and q are on the profile, and P_i and P_j do not cross. □

LEMMA 3.2. *If $x(r_j) < x(r_i)$, then P_j appears at most once in $VP(i, j)$.*

Proof. Similar to Lemma 3.1. □

We can now give a full characterization of $VP(i, j)$, for chains P_i and P_j whose x coordinate domains overlap. Assume without loss of generality that $x(l_i) < x(l_j)$. Refer to Fig. 3.

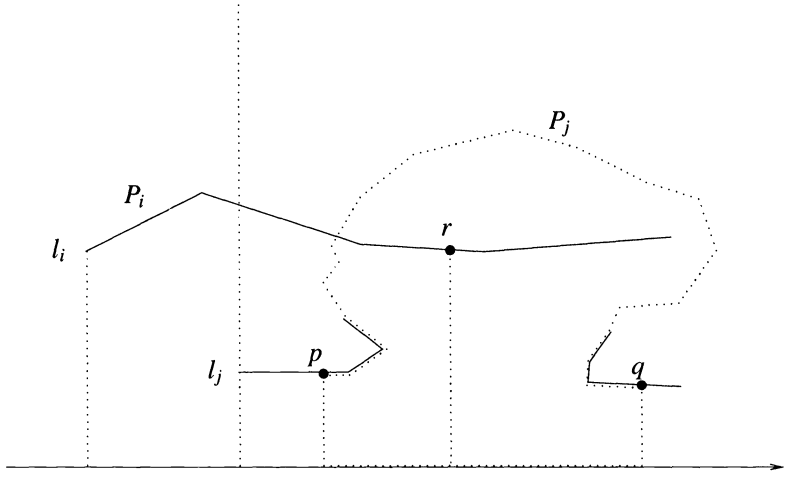


FIG. 2. *Proof of Lemma* 1.

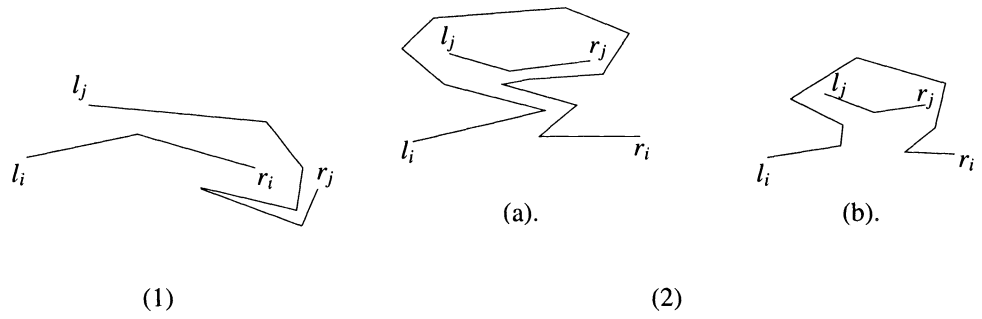


FIG. 3. *Structure of* $VP(i, j)$.

(1) If $x(r_i) < x(r_j)$, then clearly P_i and P_j each appear at least once in $VP(i, j)$, and by the previous two lemmas, each appears at most once. The profile $VP(i, j)$ therefore consists of a piece from $VP(i)$ lying left of a piece from $VP(j)$.

(2) If $x(r_j) < x(r_i)$, there are two possibilities:
   (a) Profile $VP(j)$ may lie completely above $VP(i)$, in which case $VP(i, j)$ is the single piece $VP(i)$.
   (b) If P_j appears once in $VP(i, j)$, then P_i must appear exactly twice, since pieces alternate, and the left- and rightmost pieces are from P_i.

We now state a combinatorial lemma of fundamental importance:

LEMMA 3.3. *The profile* $VP(S)$ *has* $O(|S|)$ *pieces.*

Proof. Consider the (ordered) sequence σ of indices of chains P_i that contribute pieces to $VP(S)$. There are $|S|$ different indices, and by the definition of pieces, no index i can appear twice consecutively in σ. By Lemma 3.1 (or Lemma 3.2), it is not possible to have a subsequence of the form $\ldots, i, \ldots, j, \ldots, i, \ldots, j, \ldots$. This implies that σ is a Davenport–Schinzel sequence of order 2, so its maximum length is given by $\lambda_2(|S|) = 2|S| - 1$. (See [Sh] for background on the theory of Davenport–Schinzel sequences.) □

4. An $O(n + h^2)$ algorithm. We describe now a simple $O(n + h^2)$-time algorithm for computing the visibility profile of a collection of disjoint polygons. Not only is the algorithm

relatively easy to implement, but it resolves the previously open theoretical question of whether or not an algorithm *linear* in n exists. And, even with the other results of this paper and the recent contributions of [BG] and [BC], it remains the only known algorithm linear in n that does not require linear-time triangulation.

Assume that we have indexed the chains $\mathcal{P} = \{P_1, \ldots, P_h\}$ so that their left endpoints $l_1, \ldots, l_h$ are sorted by decreasing x-coordinate. The algorithm simply considers the profiles one-by-one according to this order: step i consists of adding $VP(i)$ to $VP(1, \ldots, i-1)$ to obtain $VP(1, \ldots, i)$. The time to update the profile when we insert $VP(i)$ is linear in h and the size of P_i, implying the claimed overall time bound.

The algorithm maintains a sorted list of the leaps, $x_1, \ldots, x_K$, of the current profile $VP(1, \ldots, i-1)$. Each leap x_k in the list stores a pointer to the point $VP(1, \ldots, i-1; x_k)$. To add $VP(i)$ to the profile, we traverse $VP(i)$ to place pointers on the points $VP(i; x_1), \ldots, VP(i; x_K)$. Now for each leap x_k, we determine whether $VP(i)$ is below $VP(1, \ldots, i-1)$ at this x coordinate. (Recall that this means comparing $VP(i; x_k^-)$ to $VP(1, \ldots, i-1; x_k^-)$ and $VP(i; x_k^+)$ to $VP(1, \ldots, i-1; x_k^+)$.) If $VP(i)$ is below, we have identified a piece of $VP(i)$ in $VP(1, \ldots, i)$; we simultaneously traverse $VP(i)$ and $VP(1, \ldots, i-1)$ to the left, maintaining our pointers at the same approximate x coordinate, until we reach the x coordinate x_l, where $VP(i)$ is no longer below $VP(1, \ldots, i-1)$; x_l is the left endpoint of this piece of $VP(i)$, and consequently is a leap in $VP(1, \ldots, i)$. Similarly, we simultaneously traverse $VP(i)$ and $VP(1, \ldots, i-1)$ to the right to obtain the right endpoint, x_r. The portion of $VP(1, \ldots, i-1)$ between x_l and x_r is replaced by the corresponding portion of $VP(i)$, and the leaps at x_l and x_r are incorporated into the list, along with pointers to $VP(1, \ldots, i; x_l)$ and $VP(1, \ldots, i; x_r)$. Of course, the interval $[x_l, x_r]$ may contain leaps of $VP(1, \ldots, i-1)$ other than x_k, but this poses no difficulty to the algorithm. The special case of $x(r_i) < \min\{x(l_1), \ldots, x(l_{i-1})\}$ is handled without problem. The following lemma establishes that the new profile obtained in this manner is in fact $VP(1, \ldots, i)$, and that the updated list of leaps is the list for $VP(1, \ldots, i)$.

LEMMA 4.1. *Assume* $x(r_i) \geq \min\{x(l_1), \ldots, x(l_{i-1})\}$. *Each piece of* $VP(i)$ *in* $VP(1, \ldots, i)$ *must cover a leap of* $VP(1, \ldots, i-1)$, *that is, for each piece contributed by* $VP(i)$, *there exists a leap* x *of* $VP(1, \ldots, i-1)$ *such that* $VP(1, \ldots, i; x) \in VP(i)$.

Proof. Suppose we have a piece $[x_l, x_r]$ of $VP(i)$ in $VP(1, \ldots, i)$ that lies between consecutive leaps x_k and x_{k+1} of $VP(1, \ldots, i-1)$. Since $[x_k, x_{k+1}]$ is a single piece of $VP(1, \ldots, i-1)$, the points $VP(1, \ldots, i-1; x_k^+)$ and $VP(1, \ldots, i-1; x_{k+1}^-)$ lie on the same profile $VP(j)$. We have that $VP(j)$ lies below $VP(i)$ at x_k^+ and x_{k+1}^-, and that $VP(i)$ lies below $VP(j)$ at x_l^+, where $x_k < x_l < x_{k+1}$. But this contradicts Lemma 3.1, since $x(l_i) < x(l_j)$, by our ordering of the polygonal chains. □

We now analyze the time complexity of the algorithm. The initial indexing of the polygonal chains requires time $O(h \log h)$, to sort the left endpoints of the chains. Adding $VP(i)$ to $VP(1, \ldots, i-1)$ requires that we traverse $VP(i)$ twice—once to place pointers to $VP(i; x_1), \ldots, VP(i; x_K)$, and once during the simultaneous traversals of $VP(i)$ and $VP(1, \ldots, i-1)$. The time spent in all steps except the traversing of $VP(1, \ldots, i-1)$ is $O(|P_i| + h)$, implying a total of $O(n + h^2)$ over all steps. The simultaneous traversals of the updating step require that we traverse sections of $VP(1, \ldots, i-1)$, which consists of profiles that have already been processed. However, the portions we traverse are deleted from the current profile $VP(1, \ldots, i)$ and are never traversed again. Thus, the entire algorithm runs in time $O(n + h^2)$.

5. Lid queries. The algorithms that follow all make use of a logarithmic-time query that we call a *lid query*. A lid query is based on planar point location, a familiar notion in computational geometry. Basically, given a chain P_i and a point in the plane p, a lid query of

p on P_i asks where p is with respect to P_i. The result of such a query will give us important information about $VP(\{P_i, c\})$ for any chain c containing p such that $c \cap P_i = \emptyset$. We now describe lid queries in detail.

Consider a lid $\overline{ab}$ of $VP(i)$, the visibility profile of chain P_i. The lid $\overline{ab}$ and the subchain of P_i between a and b form a simple polygon, which we call a *pocket*, such that all points in the interior of the pocket are nonvisible from below. If a point p in the pocket lies on a chain c that does not intersect P_i, then c can be below $VP(i)$ only if it crosses $\overline{ab}$. Since $\overline{ab}$ is on $VP(i)$, crossing $\overline{ab}$ is also a sufficient condition for c to be below $VP(i)$ somewhere. We now state formally the information that we want from a *lid query*.

DEFINITION 5.1 (lid query). *For a point p not on P_i, exactly one of the following is true:*

1. *p is below $VP(i)$,*
2. *p is in a pocket of $VP(i)$,*
3. *p lies in the region above the simple, infinite chain $c = \rho_u(l_i) \cup P_i \cup \rho_u(r_i)$.*

If (1) *is true, a lid query of p on P_i returns $VP(i; x(p))$. (In Fig. 4, for example, a query on p_1 returns $VP(i; x(p_1))$.) If* 2 *is true, the lid query returns the lid $\overline{ab}$ that defines the pocket (e.g., a query on p_2 in the figure returns lid $\overline{a_2b_2}$). If* 3 *is true (as it is for p_3 in the figure), then the rays $\rho_u(l_i)$ and $\rho_u(r_i)$ together have the property that we desire in the lids, so the lid query returns them.*

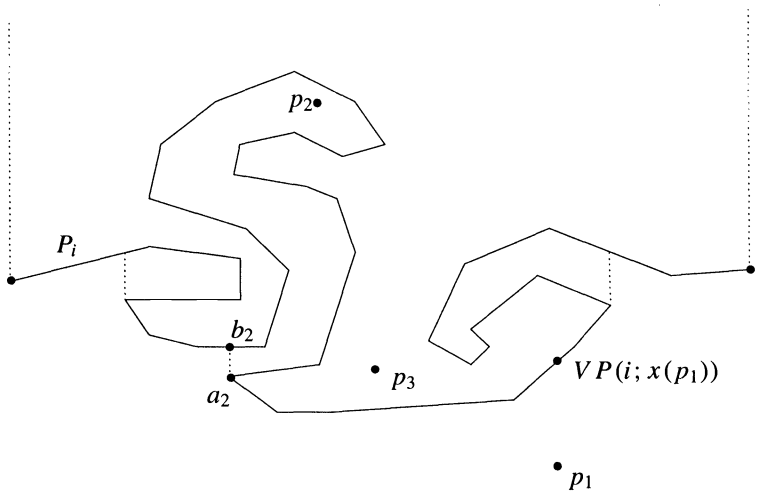


FIG. 4. *Examples of lid queries.*

Lid queries will be used often in our algorithm to determine quickly if and where a chain c containing a point p lies below the current profile $VP(S)$. Often, we will try to add a chain c to $VP(S)$ to form $VP(S \cup \{c\})$, knowing only that c contains a certain point p and that c and S are disjoint. We formalize this notion by defining the *lid property*.

DEFINITION 5.2 (lid property). *An x coordinate x has the lid property for point p and profile $VP(S)$ if, for any chain c that contains p and is disjoint from S, $VP(\{c\})$ is below $VP(S)$ somewhere only if $VP(\{c\})$ is below $VP(S)$ at x.*

When a lid query of a point p on a chain P_i returns a lid $\overline{ab}$, the x coordinate $x(a)$ satisfies the lid property for p and $VP(i)$. Our algorithm will produce, through the use of lid queries, x coordinates that satisfy the lid property for the current profile $VP(S)$; typically, these will be the x coordinates of either leaps or jumps of $VP(S)$.

Lid queries are basically planar point location. We first obtain the vertical visibility map of P_i (in $O(n)$ time for all chains, by [Ch]). Then P_i can be preprocessed to return the trapezoid of the map containing a query point in logarithmic time (by [Ki] or [EGSt], for example); this is sufficient to answer the query for case 1. For cases 2 and 3, we need to do some more work. Consider the dual graph of the trapezoidal decomposition, with the edge corresponding to the decomposition ray $\rho_u(r_i)$ deleted. This graph is a tree, and the trapezoids that comprise any given pocket correspond to a subtree. Each lid of $VP(i)$ corresponds to an edge in the tree that separates the subtree of the adjacent pocket from the rest of the tree. For each lid (including $\rho_u(l_i)$), we begin at the corresponding edge in the dual graph, and traverse through the subtree of the pocket, assigning to each node a pointer to the lid. Therefore, if the planar point location query encounters a trapezoid with a pointer to a lid, the lid query returns that lid. We see that the chains $P_1, \ldots, P_h$ can be preprocessed in $O(n)$ total time to handle lid queries in time $O(\log \bar{n})$, where $\bar{n}$ is the number of vertices on the largest of $P_1, \ldots, P_h$.

At this point we mention that our algorithms require a careful representation of a current profile $VP(S)$. We said earlier that $VP(S)$ is stored as a polygonal chain. It is important that this be done by adding pointers (along with necessary added vertices b at the top of leaps) to the original profiles $VP(i)$, $P_i \in S$. Therefore, to traverse a section of $VP(S)$, we begin by traversing a section of the appropriate profile $VP(i)$, and, upon encountering a leap at x coordinate x, leap to the new appropriate profile $VP(j)$ by means of a pointer from $VP(i; x)$ to $VP(j; x)$. We take care to mention the necessity of this representation of $VP(S)$, because our procedures will sometimes ask for a point $VP(S; x)$ by performing a lid query of $p = (x, 0)$ on the appropriate profile $VP(i)$. Since the lid query preprocessing is done on each original chain P_i, and we cannot afford to do preprocessing on an intermediate, composite profile like $VP(S)$, it is imperative that we can use a lid query on an individual chain P_i to find a point $VP(S; x)$.

6. A dynamic insertion algorithm. Through the use of lid queries, we can transform the $O(n + h^2)$ algorithm described in an earlier section into a dynamic insertion algorithm. That algorithm was not on-line, because it was necessary to order the chains in decreasing x coordinate order of their left endpoints. This ordering led to Lemma 4.1, which stated that when adding $VP(i)$ to $VP(1, \ldots, i-1)$, each piece of $VP(i)$ in $VP(1, \ldots, i)$ must cover a leap of $VP(1, \ldots, i-1)$. This meant that to construct $VP(1, \ldots, i)$, it was sufficient to compare $VP(i)$ and $VP(1, \ldots, i-1)$ only at the leaps of $VP(1, \ldots, i-1)$.

In the dynamic insertion version of the problem, we are given the chains in arbitrary order, and must compute the visibility profile of each intermediate collection of chains. Since the chains are not received in any particular order, it is possible that $VP(i)$ contributes a piece to $VP(1, \ldots, i)$ that covers no leap of $VP(1, \ldots, i-1)$. Such a piece is contained by a piece of $VP(1, \ldots, i-1)$. Therefore, the dynamic algorithm consists of checking, for each piece of $VP(1, \ldots, i-1)$, whether or not $VP(i)$ appears somewhere below it. This check can be made with the help of a lid query, as follows.

Suppose we have a piece of $VP(1, \ldots, i-1)$ that is contributed by $VP(j)$ and has left and right endpoints l and r. We first compare $VP(i)$ to $VP(j)$ at the endpoints of the piece; that is, we check if $VP(i; x(l))$ is below l or $VP(i; x(r))$ is below r. If $VP(i)$ is below at neither endpoint, then we use a lid query of the point $VP(i; x(l))$ on chain P_j. Recall that the lid $\overline{ab}$ returned by the query has the lid property, meaning that $VP(i)$ must be below $VP(j)$ at $x(a)$ if it is below anywhere. If the lid $\overline{ab}$ returned by the query has $x(l) \leq x(a) \leq x(r)$, then we compare $VP(i; x(a))$ with $\overline{ab}$. If $x(a) \notin [x(l), x(r)]$, then $VP(i)$ cannot be below $VP(j)$ anywhere in the interval $[x(l), x(r)]$, since it is not below at the endpoints of the interval, and $VP(i, j)$ has at most three pieces (Lemma 3.1). In this manner, we find a point of $VP(i)$ that is below $VP(j)$ over the interval $[x(l), x(r)]$, if one exists; we can traverse

$VP(i)$ and $VP(j)$ simultaneously to the left and to the right in order to update the profile over this interval.

We now analyze the time required to add a new chain to the current profile. The new chain P_i must be preprocessed for lid queries, in case it is queried in future steps. We must also traverse $VP(i)$ to find $VP(i; x)$ for every leap x of $VP(1, \ldots, i-1)$. There are $O(h)$ pieces of $VP(1, \ldots, i-1)$, and we may have to perform a lid query for each one. Our total time is

$$O(|P_i| + k_{i-1} + h \log \bar{n}),$$

where k_{i-1} is the number of vertices of $VP(1, \ldots, i-1)$ deleted from $VP(1, \ldots, i)$, and $\bar{n}$ is the size of the largest chain. Over h chains with n total vertices, with no chain having more than $\bar{n}$ vertices, the algorithm requires $O(n + h^2 \log \bar{n})$ time. This procedure is slightly more expensive than the $O(n + h^2)$ algorithm on which it is based, since, when we compare the new profile $VP(i)$ to a particular piece of $VP(1, \ldots, i-1)$, we must perform a lid query of $O(\log \bar{n})$ time, rather than a simple constant time check.

THEOREM 6.1. *There exists an on-line insertion algorithm for constructing the visibility profile of h chains with n vertices and $\bar{n}$ vertices on the largest chain, with time complexity $O(n + h^2 \log \bar{n})$. The insertion of an individual chain P can be performed in time $O(|P| + k + h \log \bar{n})$, where k is the size of the change.*

Our result compares favorably with the naive solution to this problem: one could perform on-line updates by simultaneously traversing $VP(i)$ and $VP(1, \ldots, i-1)$ between $x(l_i)$ and $x(r_i)$, but this approach requires $O(hn)$ time in the worst case. A slightly more sophisticated approach is to follow the algorithm that is outlined above, but, when the case of $VP(i)$ being above $VP(j)$ at both endpoints of a piece occurs, to use a simultaneous traversal instead of a lid query to update the piece. However, this approach, too, can experience worst-case performance of $O(nh)$.

7. An $O(n + h \log^2 h)$ algorithm. Using the notion of a lid query, we will describe our first $o(h^2)$ algorithm. This algorithm adds the chains one at a time to the current profile, basically by performing binary search on the profile. The algorithm is simple in all respects except the use of lid queries, and it introduces some ideas that will be used in the more sophisticated optimal algorithm.

We begin by indexing the chains $\mathcal{P} = \{P_1, \ldots, P_h\}$ so that their endpoints $l_1, \ldots, l_h$ are sorted by increasing x coordinate. (This is in contrast to the $O(n+h^2)$ algorithm, in which we wanted decreasing order.) By Lemma 3.1, a chain P_i can appear at most once in $VP(1, \ldots, i)$. In trying to add $VP(i)$ to $VP(1, \ldots, i-1)$, we will search for this solitary appearance.

The algorithm maintains the pieces of the current profile, $VP(1, \ldots, i-1)$, in a search tree. To add $VP(i)$, we perform a search. A search step at a node of the tree consists of comparing $VP(i)$ to the piece of $VP(1, \ldots, i-1)$ at that node; suppose this piece is contributed by $VP(j)$, and has endpoints l and r. If $VP(i; x(l))$ is below l, then we have found the solitary piece of $VP(i)$ in $VP(1, \ldots, i-1)$, and we can construct $VP(1, \ldots, i)$ by simultaneously traversing $VP(i)$ and $VP(1, \ldots, i-1)$ to the left and to the right. If $VP(i; x(l))$ is above l, we must determine if $VP(i)$'s contribution to $VP(1, \ldots, i)$ lies left or right of l. At this point, we perform a lid query of $VP(i; x(l))$ on P_j. If the query returns a lid $\overline{ab}$, we do the following: (1) if $x(a) < x(l)$, we continue our search on the left branch of the node, (2) if $x(r) < x(a)$, we continue on the right branch, (3) if $x(l) < x(a) < x(r)$, we simply check if $VP(i)$ is below a or crosses $\overline{ab}$. If the lid query tells us that $VP(i; x(l))$ lies in the region above $c = \rho_u(l_j) \cup P_j \cup \rho_u(r_j)$, we take the right branch, since we know that P_i does not extend as far left as $\rho_u(l_j)$. The fact that $VP(i)$ contributes at most one piece to

$VP(i, j)$ guarantees that the above search procedure will find a point of $VP(i)$ that is below $VP(1, \ldots, i-1)$, if one exists.

The procedure requires that at each step, we locate the point of $VP(i)$ with a given x coordinate. We must exhibit some care in doing this, since a naïve traversing scheme could result in some sections being covered many times, and therefore must be avoided. Instead of traversing $VP(i)$, we compute a point $VP(i, x)$ by performing a lid query of $(x, 0)$ on P_i.

To analyze the complexity of the rest of the procedure, we note that, since $VP(1, \ldots, i-1)$ has $O(h)$ pieces, the search consists of $O(\log h)$ steps. The search tree can be maintained in logarithmic time per insertion and deletion. Each search step requires a constant number of lid queries, which has time complexity $O(\log \bar{n})$, where $\bar{n}$ is the size of the largest chain of $\mathcal{P}$. Therefore the search to add $VP(i)$ requires time $O(\log h \log \bar{n})$. Of course, once we find a visible point of $VP(i)$, we must delete some of $VP(1, \ldots, i-1)$ in order to form $VP(1, \ldots, i)$, but this is $O(n)$ over all steps since no portion is deleted more than once.

By the above, the algorithm runs in time $O(n + h \log h \log \bar{n})$. We can rewrite this as $O(n + h \log^2 h)$ by the following argument. If $h = o(n/\log^2 n)$, then the entire time complexity becomes $O(n)$. If $h = \Omega(n/\log^2 n)$, then $\log h = \Omega(\log n)$, so $\log \bar{n} = O(\log h)$.

8. An optimal algorithm. We turn our attention now to a different algorithm, one which attains the optimal $\Theta(n + h \log h)$ time bound. We will describe first an algorithm that runs in time $O(n + h \log h + h \log \log h \log^2 \bar{n})$, and will then modify it to perform in optimal time.

We begin by sorting the x coordinates of the endpoints of $P_1, \ldots, P_h$, thereby obtaining a list $x_1, \ldots, x_{2h}$. If $\mathcal{P} = \{P_1, \ldots, P_h\}$, define S_1 to be the chains of $\mathcal{P}$ stabbed by the vertical line $x = x_h$. Define S_2 to be the chains of $\mathcal{P} \setminus S_1$ stabbed by $x = x_{\lfloor h/2 \rfloor}$ or $x = x_{\lfloor 3h/2 \rfloor}$. Continuing in this way, we obtain a partitioning of $\mathcal{P}$ into a class of subsets $S_1, \ldots, S_{\lceil \log 2h \rceil}$. Below, we will show that the visibility profile of a set S' of h' polygonal chains stabbed by a vertical line can be computed in time $O(n' + h' \log \bar{n}')$, where n' is the total number of vertices in S', and $\bar{n}'$ is the number of vertices on the largest chain in S'. Therefore $VP(S_1), \ldots, VP(S_{\lceil \log 2h \rceil})$ can be computed in time $O(n + h \log \bar{n})$, where $\bar{n}$ is the size of the largest chain in $\mathcal{P} = \{P_1, \ldots, P_h\}$. We will also show how to merge $VP(\mathcal{S}')$ with $VP(\mathcal{S}'')$ in $O(\bar{h} \log^2 \bar{n})$ time, for two subsets $\mathcal{S}'$ and $\mathcal{S}''$ of $\{S_1, \ldots, S_{\lceil \log 2h \rceil}\}$, with $\max\{i \mid S_i \in \mathcal{S}'\} < \min\{j \mid S_j \in \mathcal{S}''\}$, where $\bar{h}$ is the total number of polygonal chains in the sets comprising $\mathcal{S}'$ and $\mathcal{S}''$. This allows one to compute $VP(\mathcal{P})$ by recursively computing $VP(S_1 \cup \cdots \cup S_{\lceil (\log 2h)/2 \rceil})$ and $VP(S_{\lceil (\log 2h)/2 \rceil + 1} \cup \cdots \cup S_{\lceil \log 2h \rceil})$ and then merging them. Each step of the recursion requires time $O(h \log^2 \bar{n})$, and the recursion has depth $O(\log \log h)$, giving a total algorithm run-time of $O(n + h \log h + h \log \log h \log^2 \bar{n})$.

8.1. The profile for a set of stabbed polygons. We describe here the procedure for computing the visibility profile of a collection of polygonal chains $P_1, \ldots, P_h$, all of which are stabbed by a vertical line ℓ, which we take, without loss of generality, to be the y axis. We will describe a procedure for computing the portion of $VP(1, \ldots, h)$ lying to the left of ℓ; the portion lying to the right can be computed by a symmetric procedure. In this section, the notation $VP(S)$ denotes only the portion of the visibility profile of S lying to the left of ℓ, unless otherwise stated. We assume that, as a preprocessing step, we have computed $VP(i)$ and preprocessed P_i for lid queries for each chain P_i. For convenience, we will suppose that each chain P_i is oriented from its left endpoint l_i to its right endpoint r_i; we will refer to points as *preceding* or *succeeding* each other on P_i. For a chain P_i, let its *bottom point*, denoted b_i, be the point of $P \cap \ell$ with the least y coordinate; assume that we have sorted the bottom points by the y coordinate, and indexed the chains $P_1, \ldots, P_h$ so that the corresponding bottom points $b_1, \ldots, b_h$ are listed by increasing y coordinate. Note that no point of P_i succeeding b_i can lie on $VP(i)$, so it is sufficient to replace P_i by the subchain from l_i to b_i. Our procedure will inductively compute $VP(1, \ldots, k)$ by merging $VP(1, \ldots, k-1)$ with $VP(k)$. If the chains

$P_1, \ldots, P_h$ contain a total of n vertices, and no chain contains more than $\bar{n}$ vertices, then the procedure runs in time $O(n + h \log \bar{n} + h \log h)$.

The profile $VP(1, \ldots, k-1)$ consists of m pieces, which we denote, in left-to-right order, as $\pi_1, \ldots, \pi_m$. The function σ maps a piece to its chain, so that $VP(\sigma(j))$ contributes the piece π_j. Let $l(j)$ and $r(j)$ represent the left and right endpoints of the piece π_j; therefore π_j covers the x interval $[x(l(j)), x(r(j))]$.

The basic step is adding $VP(k)$ to $VP(1, \ldots, k-1)$. We will take advantage of the fact that $VP(k)$ can contribute at most one piece to $VP(1, \ldots, k)$.

LEMMA 8.1. *The chain $VP(k)$ contributes at most one piece to $VP(1, \ldots, k)$ (which is the portion of the profile lying left of the vertical stabber).*

Proof. If $k = 1$ the claim is trivial. For $k \geq 2$, it suffices to show that $VP(k)$ contributes at most one piece to the portion of $VP(j, k)$ lying left of ℓ, for all $j < k$. Because $VP(j; 0)$ is below $VP(k; 0)$, this is true by the proof of Lemma 3.3, which states that $VP(j, k)$ has at most three pieces. □

Since $VP(k)$ can contribute at most one piece, it suffices to find a single x coordinate x_k with the lid property for a point $q \in VP(k)$ and profile $VP(1, \ldots, k-1)$. The profile $VP(1, \ldots, k-1)$ consists of $O(h)$ pieces (by Lemma 3.3, since $k - 1 < h$). A natural approach would be to perform a binary search on the pieces of $VP(1, \ldots, k-1)$, asking at each step whether the desired value x_k lies left or right of the given piece. We must be able to determine efficiently whether x_k lies left or right. To do this, we must construct some structure on top of the input chains. We begin by defining, for a point $q \in VP(i)$, a *top point.*

DEFINITION 8.2. *Consider a point $q \in P_i$ on $VP(i)$. Define the* top point *of q, $t(q)$, as the first point of $P_i \cap \ell$ encountered when traversing P_i forwards from starting point q.*

As a preprocessing step, we assign to each vertex v of a profile $VP(i)$ a pointer to $t(v)$. This preprocessing can be done in linear time by traversing $VP(i)$ and P_i for each i.

We now develop a theory of *tunnels* for a chain P_i (refer to Fig. 5). Given P_i, we can construct a closed, simple curve by traversing P_i from l_i to b_i, going from b_i straight down to the origin, along the x axis, and then straight up to l_i. Denote by R_i the bounded region formed by this curve, and define R_i' to be the connected component of $R_i \cap \{(x, y) | x < 0\}$ that contains point l_i (on its boundary). We call a connected component of $R_i \setminus R_i'$ a *tunnel*, and the common boundary of R_i' with a tunnel is the *opening* of that tunnel; an opening is necessarily an interval of the y-axis. No point in the interior of a tunnel is visible from below, in the sense that a point in a tunnel of P_i is above $VP(i; x(p))$.

The intersection points of P_i with the y axis divide the y axis into a list of intervals, which we denote I_i, and can compute in linear time using Jordan sorting [HMRT]. For each interval that lies inside R_i, assign a pointer to the opening of the tunnel that contains it. For each interval outside R_i, assign a pointer to the "outside opening", namely, the interval from the highest point of $P_i \cap \ell$ to infinity. In Fig. 5, the pointers are shown as dotted lines. When we speak of the opening of a point $p = (0, y(p)) \notin P_i$ with respect to the chain P_i, we mean the opening assigned to the interval of I_i containing p. We now give the lemma that forms the basis of the binary search approach.

LEMMA 8.3. *Let π_i be a piece of $VP(1, \ldots, k-1)$, and let b_k be the bottom point of P_k. If the opening of b_k with respect to $P_{\sigma(i)}$ lies above (below) $t(l(i))$, then $VP(k)$ cannot appear in $VP(1, \ldots, k)$ to the right (left) of $x(l(i))$ unless it appears at $x(l(i))$.*

Proof. Construct a closed, simple curve c by tracing $P_{\sigma(i)}$ from $l(i)$ to $t(l(i))$, going straight down to the origin, along the x axis, and straight up to $l(i)$. Let B denote the bounded region of the plane formed by c.

Suppose the opening of b_k with respect to $P_{\sigma(i)}$ is below $t(l(i))$ (as in Fig. 6(a)). This means that p is inside $R_{\sigma(i)}$, and therefore in a tunnel (perhaps on the opening of the tunnel). The

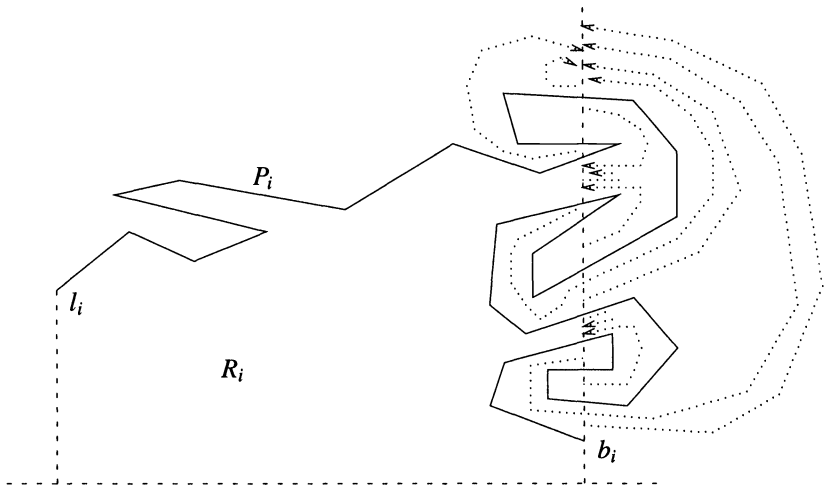


FIG. 5. *Examples of tunnels, and pointers to openings.*

chain P_k must have a point on the opening of the tunnel in order to be visible in $VP(\sigma(i), k)$, so assume we have such a point p. If P_k has a visible point of $VP(\sigma(i), k)$ lying left of $l(i)$, then it must leave the bounded region B. In order to leave B, starting at p and not intersecting $P_{\sigma(i)}$, the chain P_k must enter a tunnel with opening below $t(l(i))$, or cross $\rho_d(l(i))$. The portion of P_k in a tunnel is not visible, and P_k can exit a tunnel only through its opening, thereby re-entering B. If P_k crosses $\rho_d(l(i))$, then $VP(k)$ is below $VP(\sigma(i))$ at $x(l(i))$.

Suppose the opening is above $t(l(i))$. If b_k is outside of $R_{\sigma(i)}$, then P_k can be visible in $VP(\sigma(i), k)$ to the right of $x(l(i))$ only by crossing $\rho_d(l(i))$. If b_k is inside $R_{\sigma(i)}$ (as in Fig. 6(b)), then it is in a tunnel, and P_k can be visible only if it has point p on the opening of the tunnel. We assume P_k has a point p on this opening. To be visible in $VP(\sigma(i), k)$ to the right of $x(l(i))$, P_k must enter the bounded region B. To enter B from starting point p, P_k must enter a tunnel with opening above $t(l(i))$, or cross $\rho_d(l(i))$. While in such a tunnel, P_k is not visible, and it can exit the tunnel only through the opening above $t(l(i))$. If P_k crosses $\rho_d(l(i))$, then $VP(k)$ is below $VP(\sigma(i))$ at $x(l(i))$. □

We now describe the binary search on the pieces of $VP(1, \ldots, k-1)$ that finds x_k. At a step of the search, we have a piece π_i, and we determine the opening of b_k with respect to $P_{\sigma(i)}$; if it lies above (below) $t(l(i))$, we look for x_k to the left (right) of the piece. We compute the opening of b_k with respect to $P_{\sigma(i)}$ by maintaining, for $j = 1, \ldots, h$, the list of intervals I_j. We maintain a pointer to this list, initialized to the lowest interval, and updated after a query of b_k to the interval containing b_k. When given a new point $b_{k'}$ to query, we simply traverse through the list, starting from the pointer. In the example in Fig. 7, b' has an opening with respect to P below that of b, even though b' is above b; this is why each interval of I_j stores a pointer to its opening. Since $b_1, \ldots, b_h$ and the intervals of I_j each are ordered by increasing y coordinate, a single forwards traversal through I_j for $j = 1, \ldots, h$ allows us to perform all of these queries.

The binary search ends by isolating one piece π_i with the property that $VP(k)$ cannot appear left of $x(l(i))$ or right of $x(l(i+1)) = x(r(i))$ without appearing in the interval $[x(l(i)), x(r(i))]$. We look for a value x_k in this interval with the lid property, by performing a lid query of b_k on $P_{\sigma(i)}$. Since b_k is above $b_{\sigma(i)}$, the query will return a lid $\overline{ab}$. If the x coordinate of this lid, $x(a)$, lies in the interval $[x(l(i)), x(r(i))]$, then we set $x_k \leftarrow x(a)$, and if $x(a) <$

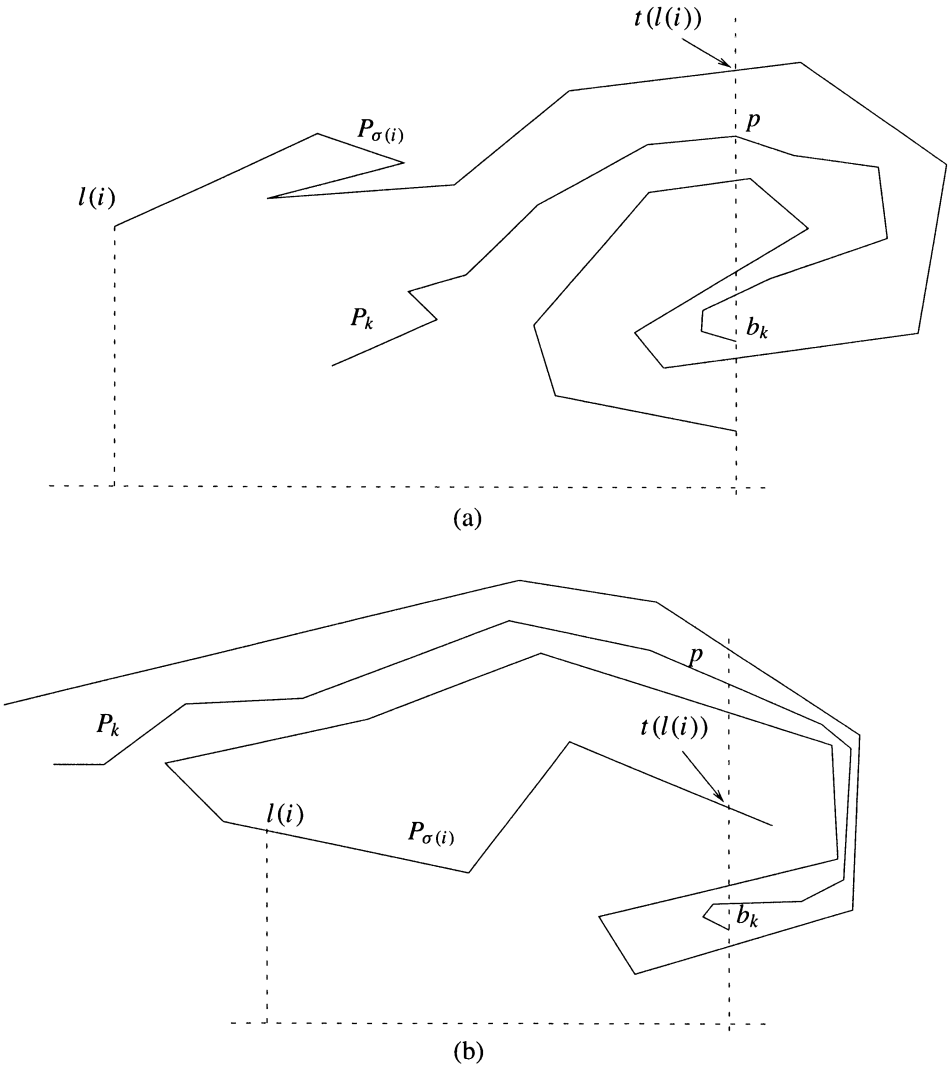


FIG. 6. *Proof of Lemma* 7.

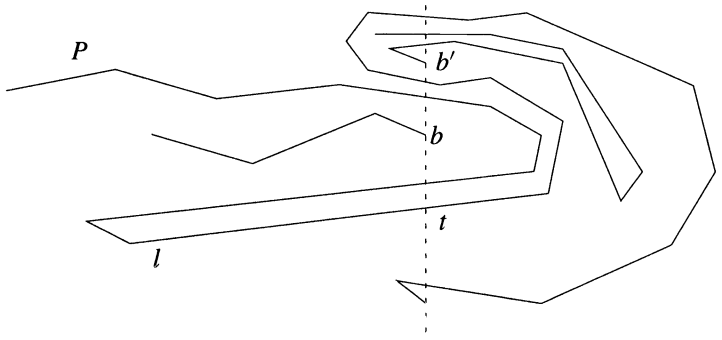


FIG. 7. *Binary search steps on a chain.*

$x(l(i))$ or $x(a) > x(r(i))$, then we set $x_k \leftarrow x(l(i))$ or $x_k \leftarrow x(r(i))$, respectively. We then compare $VP(k; x_k)$ and $VP(1, \ldots, k-1; x_k)$, and construct $VP(1, \ldots, k)$ by simultaneously traversing $VP(k)$ and $VP(1, \ldots, k-1)$. We now give the full algorithm.

Algorithm
Add P_1: We already have $VP(1)$, which consists of a single piece.
Add P_k, for $k \geq 2$:

We are given $VP(1, \ldots, k-1)$, with the pieces of $VP(1, \ldots, k-1)$ in a search tree. Perform the search described previously, using the lists I_j and the pointers to the these lists. There are two cases, which affect the way in which we construct $VP(1, \ldots, k)$:

Case 1: We find that the opening of b_k with respect to $P_{\sigma(1)}$ is above $t(l(1))$. We simply set $x_k \leftarrow x(l(1))$.

We now ask whether P_k is visible at x_k. This consists of asking whether $VP(k; x_k^-)$ is below $VP(1, \ldots, k-1; x_k^-)$, which is equivalent to asking whether P_k extends as far left as x_k^-.

If P_k is not visible at x_k, we set $VP(1, \ldots, k) \leftarrow VP(1, \ldots, k-1)$.

If P_k is visible at x_k, we compute $VP(k; x_k)$, and we traverse to the right from $VP(k; x_k)$ and $VP(1, \ldots, k-1; x_k)$ simultaneously, in order to construct the piece contributed by $VP(k)$—this gives us $VP(1, \ldots, k)$. We update the search tree of pieces of $VP(1, \ldots, k)$.

Case 2: The search returns the piece π_i.

Here we take advantage of lid queries. We query the point b_k to determine the lid $\overline{ab}$ of $VP(\sigma(i))$ that P_k must cross in order to be visible in $VP(\sigma(i), k)$. Note that since b_k is above $b_{\sigma(i)}$, the lid query will return a lid.

Subcase (a): $x(a) \in (x(l(i)), x(r(i)))$.
Set $x_k \leftarrow x(a)$.

Subcase (b): $x(a) > x(r(i))$.
Set $x_k \leftarrow x(r(i))$.

Subcase (c): $x(a) < x(l(i))$.
Set $x_k \leftarrow x(l(i))$.

We ask whether P_k is visible at x_k. This requires computing $VP(1, \ldots, k-1; x_k)$, which we already know from the lid query if $x_k \in (x(l(i)), x(r(i)))$. If x_k is $x(l(i))$ or $x(r(i))$, we can obtain $VP(1, \ldots, k-1; x_k)$ with an additional query of $P_{\sigma(i)}$ from the point $(x_k, 0)$. We are willing to spend n_k time to compute $VP(k; x_k)$, where n_k is the number of vertices of P_k, so it suffices to find $VP(k; x_k)$ by simply traversing $VP(k)$.

If $VP(k)$ is not visible at x_k, then we set $VP(1, \ldots, k) \leftarrow VP(1, \ldots, k-1)$.

If $VP(k)$ is visible at x_k, then we construct the piece contributed by $VP(k)$, by traversing simultaneously from $VP(k; x_k)$ and $VP(1, \ldots, k-1; x_k)$, both left and right—this gives $VP(1, \ldots, k)$. We update the search tree of pieces of $VP(1, \ldots, k)$.
End Algorithm

We now show that the value x_k produced by a step of the algorithm has the lid property.

LEMMA 8.4. *If $VP(k)$ appears in $VP(1, \ldots, k)$, then it appears in $VP(1, \ldots, k)$ at x_k.*

Proof. The claim follows directly from Lemma 8.3 for Case 1 of the algorithm.

For Case 2, it follows from Lemma 8.3 that if $VP(k)$ appears in $VP(1, \ldots, k)$, then it appears in the interval $[x(l(i)), x(r(i))]$.

Suppose the query returns a lid $\overline{ab}$ such that $x(a) \in (x(l(i)), x(r(i)))$. Then $VP(k)$ is somewhere below $VP(\sigma(i))$ only if P_k crosses $\overline{ab}$. Thus, $VP(k)$ appears in $VP(1, \ldots, k)$

only if $VP(k)$ is below $VP(1, \ldots, k-1)$ at $x(a)$, implying that $x_k = x(a)$ satisfies the lid property.

Suppose the query returns a lid $\overline{ab}$ such that $x(a) > x(r(i))$. We know that if $VP(k)$ appears in $VP(\sigma(i), k)$, then it appears at $x(a)$. Since $y(b_{\sigma(i)}) < y(b_k)$, $VP(k)$ can contribute at most one piece to the portion of $VP(\sigma(i), k)$ lying left of the y axis. Therefore, if $VP(k)$ is below $VP(\sigma(i))$ at $x(a)$ and at a value $x \in [x(l(i)), x(r(i))]$, then it is below at $x_k = x(r(i))$, which implies that x_k satisfies the lid property. A similar argument handles the case where $x(a) < x(l(i))$. □

The entire procedure for computing $VP(1, \ldots, h)$ requires time $O(n + h \log \bar{n} + h \log h)$, where n is the total number of vertices on $P_1, \ldots, P_h$ and $\bar{n}$ is the number of vertices on the largest chain. We established earlier that all preprocessing can be performed in $O(n)$ time. The bottom points are sorted by y coordinate in $O(h \log h)$ time.

The basic step of the procedure consists of merging $VP(k)$ with $VP(1, \ldots, k-1)$. We perform a binary search on the list of pieces of $VP(1, \ldots, k-1)$. The cost of each search step is a constant plus the number of intervals of the list $I_{\sigma(i)}$ that are traversed. The constant cost is charged to the search step, and the cost of each interval is charged to the interval itself, since no interval is traversed more than once. Since $VP(1, \ldots, k-1)$ has $O(h)$ pieces, all steps of all searches require $O(n + h \log h)$ time.

To merge $VP(k)$ with $VP(1, \ldots, k-1)$, we perform a constant number of lid queries, each in time $O(\log \bar{n})$, for a total of $O(h \log \bar{n})$ over all steps. Each step also traverses some portion of $VP(k)$, in time linear in the size of $VP(k)$, for a total of $O(n)$ time. Also, step k traverses the portion of $VP(1, \ldots, k-1)$ that is deleted from $VP(1, \ldots, k)$; all of these deletions require $O(n)$ time. This establishes the time bound of $O(n + h \log \bar{n} + h \log h)$ on the procedure.

8.2. Merging. We describe the merging of the visibility profiles of two subsets $\mathcal{S}'$ and $\mathcal{S}''$ of $\mathcal{S} = \{S_1, \ldots, S_{\lceil \log h \rceil}\}$, where $\max\{i \mid S_i \in \mathcal{S}'\} < \min\{j \mid S_j \in \mathcal{S}''\}$. We have a family of vertical lines such that every chain in $\mathcal{S}'$ is stabbed by at least one of the lines, but no chain of $\mathcal{S}''$ is stabbed by a line. Since the vertical lines separate the elements of $\mathcal{S}''$, we can individually consider the interval between each pair of consecutive lines. Therefore, we consider a vertical strip bordered by the lines π_l and π_r. All chains of $\mathcal{S}'$ that appear in $VP(\mathcal{S}')$ in the strip are stabbed by either π_l or π_r, and no chain of $\mathcal{S}''$ appearing in $VP(\mathcal{S}'')$ is stabbed by either line.

Inductively, we assume that we have, for $VP(\mathcal{S}')$ ($VP(\mathcal{S}'')$) over the strip, a sorted list of all leaps, and for each leap x of $VP(\mathcal{S}')$ ($VP(\mathcal{S}'')$), a pointer to $VP(\mathcal{S}'; x)$ ($VP(\mathcal{S}''; x)$). We merge the lists to form a single sorted list $x_1, \ldots, x_K$ of all leaps in $VP(\mathcal{S}')$ and $VP(\mathcal{S}'')$. For a leap x_k from $VP(\mathcal{S}')$ ($VP(\mathcal{S}'')$), we must compute a pointer to $VP(\mathcal{S}''; x_k)$ ($VP(\mathcal{S}'; x_k)$). We do this through lid queries, as follows. We can assume that in forming the list $x_1, \ldots, x_K$, every leap x_k from $VP(\mathcal{S}')$ knows which profile from $\mathcal{S}''$ contributes $VP(\mathcal{S}''; x_k)$. (This consists of knowing the leaps from $VP(\mathcal{S}'')$ that are nearest to x_k to the left and right.) Since we know the profile $VP(j)$ that contributes $VP(\mathcal{S}''; x_k)$, we can compute $VP(j; x_k) = VP(\mathcal{S}''; x_k)$ by querying the point $(x_i, 0)$ on P_j. Since $(x_k, 0)$ lies below $VP(j)$ (by our assumption that chains lie above the x axis), the query returns $VP(j; x_k)$.

We define a *subpiece* of $VP(\mathcal{S}')$ or $VP(\mathcal{S}'')$ as the portion of the profile between two consecutive leaps in the merged list $x_1, \ldots, x_K$. Note that a subpiece is a subset of some piece. A trivial special case occurs if one of the subpieces is at infinity. We will concentrate on the nontrivial situation, where the two subpieces are contributed by two profiles, $VP(i)$ and $VP(j)$. The following lemma motivates the merge procedure.

LEMMA 8.5. *Suppose we have a subpiece over the interval* $[x_k, x_{k+1}]$ *such that* $VP(i)$ *and* $VP(j)$ *contribute this subpiece to* $VP(\mathcal{S}')$ *and* $VP(\mathcal{S}'')$, *respectively. Then* $VP(\mathcal{S}')$ *is below* $VP(\mathcal{S}'')$ *somewhere in* $[x_k, x_{k+1}]$ *only if* $VP(i)$ *is below* $VP(j)$ *at* x_k *or* x_{k+1}.

Proof. We know that $VP(j)$, as an element of $VP(\mathcal{S}'')$, does not intersect π_l nor π_r, whereas $VP(i)$, as an element of $VP(\mathcal{S}')$, does intersect at least one of the two. This implies that either $x(l_i) < x(l_j)$ or $x(r_i) > x(r_j)$, which, by Lemma 3.1 or Lemma 3.2, means that $VP(j)$ appears at most once in $VP(i, j)$. This means it is impossible for $VP(j)$ to be below $VP(i)$ at both x_k and x_{k+1} when $VP(i)$ is below $VP(j)$ somewhere in between x_k and x_{k+1}. □

Now we combine the two profiles over the subpiece $[x_k, x_{k+1}]$. If $VP(j)$ is below $VP(i)$ at both x_k and x_{k+1}, then the entire subpiece is contributed by $VP(j)$. Below we describe a procedure for the case when $VP(i)$ is below at one of x_k and x_{k+1}, and $VP(j)$ is below at the other. We then show how this procedure can be modified to handle the case where $VP(i)$ is below at both x_k and x_{k+1}.

If $VP(i)$ is below at one of x_k and x_{k+1}, and $VP(j)$ is below at the other, then our task consists of finding the unique leap in $VP(i, j)$ between x_k and x_{k+1}, without traversing portions that are part of $VP(\mathcal{S}' \cup \mathcal{S}'')$. A naïve scheme could take time linear in the size of the portions of $VP(\mathcal{S}')$ and $VP(\mathcal{S}'')$ between x_k and x_{k+1}, but our approach requires only polylog time. Assume that the vertices of each original profile have been numbered in left-to-right order, and placed in a data structure so that if we are given a pointer to a vertex of the profile, we can in constant time return the numbering of the vertex. Our procedure maintains two pointers to $VP(i)$, denoted p_l^i and p_r^i, which are initialized to $VP(i; x_k)$ and $VP(i; x_{k+1})$, and pointers p_l^j and p_r^j to $VP(j)$, initialized to $VP(j; x_k)$ and $VP(j; x_{k+1})$. The pointers p_l^i and p_l^j will be maintained at the same x-coordinate, as will p_r^i and p_r^j. Initially we know that there is exactly one leap of $VP(i, j)$ between $x(p_l^i) = x(p_l^j)$ and $x(p_r^i) = x(p_r^j)$; the procedure maintains this property while moving $x(p_l^i) = x(p_l^j)$ and $x(p_r^i) = x(p_r^j)$ closer together, eventually sandwiching the leap. The procedure alternates steps on the pointer pairs (p_l^i, p_r^i) and (p_l^j, p_r^j). We describe a step on the pair (p_l^i, p_r^i):

1. Query the numbering of the vertices of $VP(i)$ nearest p_l^i and p_r^i, and assign these numberings to p_l^i and p_r^i.
2. Find q, the vertex of $VP(i)$ whose numbering is halfway between the numberings of p_l^i and p_r^i.
3. Compute $VP(j; x(q))$.
4. Compare the y coordinates of $VP(j; x(q))$ and $q = VP(i; x(q))$; this tells us whether the leap is left or right of $x(q)$; accordingly, update either p_l^i and p_l^j or p_r^i and p_r^j.

Upon completion of this step on the pair (p_l^i, p_r^i), perform a symmetric step on (p_l^j, p_r^j), and continue to alternate the steps. Eventually the total number of vertices on $VP(i)$ between p_l^i and p_r^i and on $VP(j)$ between p_l^j and p_r^j is less than a small, preset constant, so in constant time we find the leap and complete this subpiece of $VP(\mathcal{S}' \cup \mathcal{S}'')$.

A slight modification of the above procedure handles the case where $VP(i)$ is below $VP(j)$ at both x_k and x_{k+1}. Query $VP(j; x_k)$ on $VP(i)$, to find a lid $\overline{ab}$ which $VP(j)$ must cross in order to contribute a piece to $VP(i, j)$. If $x(a) \notin [x_k, x_{k+1}]$, then $VP(j)$ is not below $VP(i)$ anywhere in the interval $[x_k, x_{k+1}]$. If $x(a) \in [x_k, x_{k+1}]$, then perform a lid query on the point $(x(a), 0)$ to compute $VP(j; x(a))$. If $VP(i)$ is below $VP(j)$ at $x(a)$, then the subpiece between x_k and x_{k+1} is contributed totally by $VP(i)$; otherwise, we break the subpiece $[x_k, x_{k+1}]$ into two subpieces, $[x_k, x(a)]$ and $[x(a), x_{k+1}]$, and process each subpiece with the above procedure.

Consider the total time of merging $VP(\mathcal{S}')$ and $VP(\mathcal{S}'')$. Let $\bar{h}$ represent the total number of chains in $\mathcal{S}'$ and $\mathcal{S}''$, and $\bar{n}$ the number of vertices on the largest chain in the set $\mathcal{S}' \cup \mathcal{S}''$. Creating the combined sorted lists of leaps $x_1, \ldots, x_K$ for all strips takes time $O(\bar{h})$, because we already have the sorted lists of leaps for $\mathcal{S}'$ and $\mathcal{S}''$ separately. Computing $VP(i; x_k)$ and

$VP(j; x_k)$ for every leap x_k requires time $O(\bar{h} \log \bar{n})$, since it consists of performing one lid query per leap. We then process each of the $O(\bar{h})$ subpieces separately, perhaps breaking some subpieces into two subpieces with the help of a single lid query. Processing a subpiece consists of alternating steps on the pairs of pointers (p_l^i, p_r^i) and (p_l^j, p_r^j). Each step consists of one lid query plus some constant time work, and is therefore $O(\log \bar{n})$. Because every two steps eliminate at least half of the vertices of $VP(i)$ between p_l^i and p_r^i and of $VP(j)$ between p_l^j and p_r^j, the number of steps is $O(\log \bar{n})$. Therefore each subpiece requires $O(\log^2 \bar{n})$ time, for a total of $O(\bar{h} \log^2 \bar{n})$ time to merge $VP(\mathcal{S}')$ and $VP(\mathcal{S}'')$.

8.3. Putting it together: An optimal algorithm. The §§8.1 and 8.2 describe how to compute $VP(\mathcal{P})$ for $\mathcal{P} = \{P_1, \ldots, P_h\}$ in time $O(n + h \log h + h \log \log h \log^2 \bar{n})$, where n is the total number of vertices in $\mathcal{P}$, and $\bar{n}$ is the number of vertices on the largest chain of $\mathcal{P}$. A modification allows this algorithm to compute $VP(\mathcal{P})$ in optimal $\Theta(n + h \log h)$ time. The modification consists of breaking $\mathcal{P}$ into two groups, the "large" chains and the "small" ones, computing the visibility profile of each group separately, and then merging the profiles with a final linear-time merge.

The first observation to be made is that if $h = O(n/\log^3 n)$, then the algorithm's complexity is $O(n)$. Motivated by this observation, we break $\mathcal{P}$ into two groups as follows: all chains in $\mathcal{P}$ with more than $\log^3 n$ vertices are placed in the "large" group, and the rest in the "small" group. The large group can have no more than $n/\log^3 n$ members, so the algorithm can compute the visibility profile of this group in $O(n)$ time.

Assume that the small group has $h = \Omega(n/\log^3 n)$ members. (If not, the algorithm is $O(n)$ on this group.) No chain of the small group has more than $\log^3 n$ vertices, implying that $\bar{n} \leq \log^3 n$. The complexity of the algorithm is therefore $O(n + h \log h + h \log \log h \log^2(\log^3 n)) = O(n + h \log h + h \log \log h (\log \log n)^2)$. Since $h = \Omega(n/\log^3 n)$, we have that $\log \log n = O(\log \log h)$, which gives a complexity of $O(n + h \log h + h(\log \log h)^3) = O(n + h \log h)$. Therefore, the visibility profiles of both the small group and the large group can be computed in $O(n + h \log h)$, which gives $VP(\mathcal{P})$ in the same time bound.

9. Conclusion. In this paper, we have given a number of algorithms for computing visibility polygons in a polygonal domain, which culminated in a time-optimal $\Theta(n + h \log h)$ result. Another result was a simple $O(n + h^2)$-time algorithm, which is notable as the only known algorithm whose time-complexity is linear in n that does not require linear-time triangulation. We also gave a procedure that allows dynamic insertion of the polygonal holes.

Our visibility profile results immediately give new bounds on computing convex hulls of disjoint simple polygons, since the lower hull of a set of obstacles can be extracted in linear ($O(n)$) time from the lower envelope of the obstacles.

Several open problems present themselves:

- Can one develop a dynamic algorithm that allows both insertion and deletion of the holes?
- Does there exist an $O(n + f(n))$-time algorithm that does not require linear-time triangulation, where $f(h) = o(h^2)$?
- Can we extend our results to the case of (weak) visibility from a line segment?
- What can be done in the case that the h polygons are not necessarily disjoint, but have up to k intersection points?
- How quickly can one compute the visibility profile of h simple chains that are *not* assumed to be pairwise-disjoint? If, for example, there are k crossing points in the arrangement of chains, can we compute the visibility profile (lower envelope) in time $O(n + k + h \log h)$, or something close to this? (This problem is suggested to us by Reuven Bar Yehuda.)

REFERENCES

[AM] E. ARKIN AND J. S. B. MITCHELL, *An optimal visibility algorithm for a simple polygon with star-shaped holes*, technical report 746, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, NY, June, 1987.

[As] T. ASANO, *Efficient algorithms for finding the visibility polygons for a polygonal region with holes*, Trans. of IECE of Japan, E-68 (1985), pp. 557–559.

[AAGHI] T. ASANO, L. GUIBAS, J. HERSHBERGER, AND H. IMAI, *Visibility of disjoint polygons*, Algorithmica, 1 (1986), pp. 49–63.

[BC] R. BAR-YEHUDA AND B. CHAZELLE, *Triangulating a set of non-intersecting and simple polygonal chains*, manuscript, 1992.

[BG] R. BAR-YEHUDA AND R. GRINWALD, *An $O(n+h^{1+\epsilon})$-time algorithm to merge h simple polygons*, tech. report 657, Computer Science Dept., Technion—Israel Institute of Technology, Haifa, Israel, November, 1990.

[Ch] B. CHAZELLE, *Triangulating a simple polygon in linear time*, Discrete Comput. Geom., 6 (1991), pp. 485–524.

[CTV] K. CLARKSON, R. E. TARJAN, AND C. VAN WYK, *A fast Las Vegas algorithm for triangulating a simple polygon*, Discrete Comput. Geom., 4 (1989), pp. 423–432.

[EGSt] H. EDELSBRUNNER, L. J. GUIBAS, AND J. STOLFI, *Optimal point location in a monotone subdivision*, SIAM J. Comput., 15 (1986), pp. 317–340.

[EA] H. A. EL GINDY AND D. AVIS, *A linear algorithm for computing the visibility polygon from a point*, J. Algorithms, 2 (1981), pp. 186–197.

[HMRT] K. HOFFMAN, K. MEHLHORN, P. ROSENSTIEHL, AND R. TARJAN, *Sorting Jordan sequences in linear time using level-linked search trees*, Inform. and Control, 68 (1986), pp. 170–184.

[JS] B. JOE AND R. B. SIMPSON, *Correction to Lee's visibility polygon algorithm*, BIT, 27 (1987), pp. 458–473.

[Ki] D. G. KIRKPATRICK, *Optimal search in planar subdivisions*, SIAM J. Comput., 12 (1983), pp. 28–35.

[KKT] D. G. KIRKPATRICK, M. M. KLAWE, AND R. E. TARJAN, *Polygon triangulation in $O(n \log\log n)$ time with simple data-structures*, Proc. Sixth Annual ACM Symposium on Computational Geometry, Berkeley, CA, June 6–8, 1990, Assoc. for Computing Machinery, New York, NY, pp. 34–43.

[Le1] D. T. LEE, "Proximity and Reachability in the Plane," Ph.D. Thesis, Report R-831, Dept. of Electrical Engineering, University of Illinois at Urbana-Champaign, Nov. 1978.

[Le2] ———, *Visibility of a simple polygon*, Comput. Vision, Graphics, and Image Processing, 22 (1983), pp. 207–221.

[O'R] J. O'ROURKE, *Art Gallery Theorems and Algorithms*, Oxford University Press, Cambridge, UK, 1987.

[Se] R. SEIDEL, *A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons*, Comput. Geom. Theory Appl., 1 (1991), pp. 51–64.

[Sh] M. SHARIR, *Davenport–Schinzel Sequences and their Geometric Applications*, pp. 253–278, NATO ASI Series, Vol. F40, Theoretical Foundations of Computer Graphics and CAD, R.A. Earnshaw, ed., Springer-Verlag, Berlin, Heidelberg, 1988.

[SO] S. SURI AND J. O'ROURKE, *Worst-case optimal algorithms for constructing visibility polygons with holes*, Proc. Second Annual ACM Symposium on Computational Geometry, Yorktown Heights, NY, June 1986, Assoc. for Computing Machinery, New York, NY, pp. 14–23.

[TV] R. E. TARJAN AND C. VAN WYK, *An $O(n \log\log n)$-time algorithm for triangulating a simple polygon*, SIAM J. Comput., 17 (1988), pp. 143–178.

SIAM J. COMPUT.
Vol. 24, No. 2, pp. 203–226, April 1995

A RANDOMIZED MAXIMUM-FLOW ALGORITHM*

JOSEPH CHERIYAN† AND TORBEN HAGERUP‡

Abstract. A randomized algorithm for computing a maximum flow is presented. For an n-vertex m-edge network, the running time is $O(nm + n^2(\log n)^2)$ with probability at least $1 - 2^{-\sqrt{nm}}$. The algorithm is always correct, and in the worst case runs in $O(nm \log n)$ time. The only use of randomization is to randomly permute the adjacency lists of the network vertices at the start of the execution.

The analysis introduces the notion of premature target relabeling (PTR) events and shows that each PTR event contributes $O(\log n)$ amortized time to the overall running time. The number of PTR events is always $O(nm)$; however, it is shown that when the adjacency lists are randomly permuted, then this quantity is $O(n^{3/2}m^{1/2} + n^2 \log n)$ with high probability.

Key words. maximum flow, randomized algorithm, random permutations, scaling, dynamic tree, Fibonacci heap

AMS subject classifications. 68Q20, 68Q25, 68R05, 90C35

1. Introduction.

A *network* is a graph together with one or more functions from the edges to the real numbers. Problems on networks arise often in theory and in practice. One of the central problems in this area is that of finding a *maximum flow* in a network. The input to the problem consists of a graph with two distinguished vertices, the *source* and the *sink*, and a nonnegative function on the edges called the *capacity*. Let n and m denote the number of vertices and edges, respectively, and let U denote the maximum capacity of any edge.

The first algorithm for the problem was presented by Ford and Fulkerson [FF57], [FF62]. Their algorithm does not run in polynomial time; moreover, when the capacities are irrational, it may not even terminate. More than a decade later, Edmonds and Karp [EK72] proposed a modification of the algorithm of [FF57] that runs in polynomial time. Independently, Dinic [D70] gave a faster polynomial-time algorithm.

The running times of the Dinic and Edmonds–Karp algorithms can be bounded by functions that depend only on n and m, but not on the actual edge capacities. This behavior is considered to be attractive and has been studied in the more general setting of combinatorial optimization. The input to a combinatorial optimization problem consists of a discrete structure (e.g., the graph in the maximum-flow problem), together with numeric parameters (e.g., the edge capacities in the maximum-flow problem). For an instance of the problem, let p denote the size of (a reasonable encoding of) the discrete structure, and let ℓ denote the maximum size of any numeric parameter (e.g., if the numeric parameters are integers with absolute values bounded by U, then one can take $\ell = 1 + \lceil \log(U+1) \rceil$; see Grötschel, Lovász, and Schrijver [GLS88]). An algorithm is said to be *strongly polynomial* if its running time in the arithmetic model (i.e., the total number of arithmetic operations, comparisons, and data transfers executed) is polynomial in p, and the maximum size of any number computed by the algorithm is polynomial in $p\ell$. The maximum-flow algorithms in [D70] and [EK72] are strongly polynomial.

*Received by the editors November 1, 1991; accepted for publication (in revised form) October 21, 1993. A preliminary and abridged version of this paper was presented at the 30th IEEE Symposium on Foundations of Computer Science in October 1989. Most of the research was carried out while both authors were at the Fachbereich Informatik of the Universität des Saarlandes, Saarbrücken, Germany.

†Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Ontario, N2L 3G1, Canada. This research was supported in part by the ESPRIT II Basic Research Actions Program of the European Community under contract no. 3075 (project ALCOM) and by the National Science Foundation, the Air Force Office of Scientific Research, and the Office of Naval Research, through National Science Foundation grant DMS-8920550.

‡Max-Planck-Institut für Informatik, D-66123 Saarbrücken, Germany. This research was supported in part by the ESPRIT II Basic Research Actions Program of the European Community under contract no. 3075 (project ALCOM).

The publication of [D70] and [EK72] led to further research on maximum-flow algorithms. Karzanov [K74] introduced the notion of a *preflow*, a generalization of a flow, and gave an algorithm that computes a maximum flow in $O(n^3)$ time by manipulating a preflow. Other improvements in the time complexity followed, and this line of research culminated in [ST83], where Sleator and Tarjan introduced a new data structure for dynamic trees and showed that by using this data structure, a maximum flow can be computed in $O(nm \log n)$ time. For graphs with $O(n^2/(\log n)^3)$ edges, this running time was the fastest then known. For networks with integer capacities, Gabow [Ga85] subsequently reported a simple algorithm based on the scaling technique with a running time of $O(nm \log U)$; under the so-called *similarity assumption*, i.e., $U = n^{O(1)}$, this algorithm is as fast as the one in [ST83].

A new approach for computing maximum flows was presented by Goldberg and Tarjan [GT88], based on earlier work by Goldberg [G85]. [GT88] introduced a generic algorithm for computing a maximum flow that works by manipulating a preflow, and gave a specific algorithm that uses the dynamic trees data structure and runs in time $O(nm \log(n^2/m))$. This improves on the algorithm in [ST83] for relatively dense graphs with $m = n^{2-o(1)}$. Cheriyan and Maheshwari [CM89] investigated several specific instances of the generic algorithm and showed that one of them runs in $\Theta(n^2\sqrt{m})$ time. For the special case of integer capacities, Ahuja and Orlin [AO89] devised an algorithm based on the scaling technique, called the *excess scaling algorithm*, that runs in $O(nm + n^2 \log U)$ time. Ahuja, Orlin, and Tarjan [AOT89] obtained several fast algorithms for this case by combining the dynamic trees data structure with the excess scaling algorithm.

At a somewhat general level, several aspects of the present paper were influenced by recent research on the *minimum-cost flow problem*. Edmonds and Karp [EK72], introducing the scaling technique, gave a polynomial minimum-cost flow algorithm for the special case of integer capacities. The algorithm, however, is not strongly polynomial. Edmonds and Karp left open the challenging question of whether a strongly polynomial algorithm exists. Although this question attracted research interest, no significant progress was reported for over a decade, until Tardos [T85] answered the question in the affirmative. There followed a number of other strongly polynomial algorithms: Fujishige [F86], Galil and Tardos [GaT88], Goldberg and Tarjan [GT89], [GT90], and Orlin [O88]. An idea common to these papers was the use of the scaling method for designing efficient strongly polynomial algorithms, and the most efficient realization of this idea was given in [O88], where the Edmonds–Karp capacity-scaling algorithm was "adapted" to real-valued capacities in order to achieve the fastest strongly polynomial running time known. The design of our algorithm was also motivated by this idea.

The focus of our work is on using randomization to improve on the Sleator–Tarjan time bound of $O(nm \log n)$ for finding a maximum flow. We show that by suitably incorporating the dynamic trees data structure into the excess scaling algorithm *and* randomly permuting the adjacency lists of the network vertices at the start of the execution, we can compute a maximum flow in $O(nm + n^2(\log n)^2)$ time with high probability, and in $O(nm \log n)$ time in the worst case.

The running time analysis introduces the notion of *premature target relabeling* (PTR) events and shows that each PTR event contributes $O(\log n)$ amortized time to the overall running time. The number of PTR events is always $O(nm)$; however, it is shown that when the adjacency lists are randomly permuted, then this quantity is $O(n^{3/2}m^{1/2} + n^2 \log n)$ with high probability.

Section 3 discusses a variant of the excess scaling algorithm, and in §4 an efficient strongly polynomial algorithm is designed, based on the algorithm in §3. The running time is analyzed in §§5–8. The critical component of the running time is identified in §5, and it is shown that the algorithm is strongly polynomial. Section 6 introduces PTR events and gives a probabilistic

bound on the number of these events, assuming that the adjacency lists are permuted randomly. The analysis is completed in §§7 and 8, first using simple arguments that lead to a somewhat loose time bound, and then using a more refined argument that gives the tightest bound known on the running time. Some conclusions are presented in §9.

2. Preliminaries. We use the traditional model for the study of problems on networks. Capacities and flow values are represented by real numbers, and all other quantities are represented by integers. For n-vertex input networks, we allow integers of absolute value $n^{O(1)}$, and we charge constant time for each arithmetic operation (i.e., addition, subtraction, comparison, or division by 2) on real numbers or integers, and for each data transfer. Further, we charge constant time for generating one random integer.

A *flow network* consists of a directed graph $G = (V, E)$, assumed to be symmetric (i.e., $(v, w) \in E \iff (w, v) \in E$), and a function $cap : E \to \mathbb{R}_+ \cup \{0\}$, the *capacity*, together with two distinguished vertices, the *source*, s, and the *sink*, t. Let $n = |V|$ and $m = |E|$, and let U denote the maximum capacity of any edge. We assume that $m \geq n \geq 3$ and that $U > 0$. For $v \in V$, let $\Gamma(v) = \{w \in V : (v, w) \in E\} = \{u \in V : (u, v) \in E\}$, and for every $(v, w) \in V \times V$, let $tail(v, w) = v$ and $head(v, w) = w$.

A *preflow* in G is a function $f : E \to \mathbb{R}$ with the following properties:

(1) $f(w, v) = -f(v, w)$ for all $(v, w) \in E$ (antisymmetry constraint);

(2) $f(v, w) \leq cap(v, w)$ for all $(v, w) \in E$ (capacity constraint);

(3) $\sum_{u \in \Gamma(v)} f(u, v) \geq 0$ for all $v \in V - \{s\}$ (nonnegativity constraint).

A preflow f in G is a *flow* if $\sum_{u \in \Gamma(v)} f(u, v) = 0$ for all $v \in V - \{s, t\}$ (flow conservation constraint). The *value* of f is $\sum_{u \in \Gamma(t)} f(u, t)$, and a *maximum flow* in G is a flow in G of maximum value.

For a fixed preflow f under consideration and for every vertex $v \in V$, the *flow excess* of v, $e(v)$, is defined as $\sum_{u \in \Gamma(v)} f(u, v)$, i.e., the net flow into v. A vertex v is called *active* if $v \in V - \{s, t\}$ and $e(v) > 0$.

An edge $(v, w) \in E$ is *residual* (with respect to f) if $f(v, w) < cap(v, w)$. The *residual capacity* of an edge (v, w), $rescap(v, w)$, is defined to be $cap(v, w) - f(v, w)$.

A *labeling* of G is a function $d : V \to \mathbb{N}_0 = \{0, 1, 2, \ldots\}$. The labeling is *valid* for G and a preflow f in G exactly if $d(t) = 0$, $d(s) = n$, and $d(v) \leq d(w) + 1$ for each residual edge (v, w). We call a residual edge (v, w) *eligible* if $d(v) = d(w) + 1$.

We now describe a generic algorithm that uses a preflow f in G and a labeling d of G. The basic procedure for manipulating f is as follows:

procedure *push*$((v, w) : edge; c : real)$;
  precondition: v is active, (v, w) is eligible and $0 < c \leq \min\{e(v), rescap(v, w)\}$.
    $f(v, w) := f(v, w) + c$; $f(w, v) := f(w, v) - c$;
    $e(v) := e(v) - c$; $e(w) := e(w) + c$;

By a *push* over (v, w) of *value* c we mean an execution of *push* with parameters (v, w) and c. v and w are called the *tail vertex* and the *head vertex* of the push, respectively. The push is called *saturating* if $f(v, w) = cap(v, w)$ afterwards; otherwise it is called *nonsaturating*. For a vertex v, a push *out of* v is a push over any edge of the form (v, w), and a push *into* v is a push over any edge of the form (u, v).

The basic procedure for manipulating the labeling d is as follows:

procedure *relabel*$(v : vertex)$;
  precondition: v is active, and no edge (v, w) emanating from v is eligible.
    $d(v) := d(v) + 1$;

An execution of *relabel*(v) is called a *relabeling* of v.

The generic algorithm starts by setting the flow on each edge that leaves the source equal to its capacity and the flow on each edge that is not incident with the source equal to zero. It also fixes some initial valid labeling. The algorithm then repeatedly executes *push* and *relabel* operations in any order. When no *push* or *relabel* operation has its precondition satisfied, i.e., when there are no more active vertices, the algorithm terminates. A formal description follows.

procedure *generic initialize*;
  for all $(v, w) \in E$ **do** $f(v, w) := 0$;
  for all $v \in V$ **do** $e(v) := 0$;
  for all $(s, v) \in E$ **do**
  begin
    $f(s, v) := cap(s, v)$;
    $f(v, s) := -f(s, v)$;
    $e(v) := cap(s, v)$;
  end;
  for all $v \in V - \{s\}$ **do** $d(v) := 0$;
  $d(s) := n$;

The generic maximum-flow algorithm:
  generic initialize;
  while there is an active vertex **do**
    execute some *push* or *relabel* operation whose precondition is satisfied.

There are some minor differences between the algorithm above and the generic maximum-flow algorithm of Goldberg and Tarjan [GT88]: the *push*(v, w) operation of [GT88] always sends $c = \min\{e(v), rescap(v, w)\}$ units of flow, and the *relabel*(v) operation of [GT88] sets $d(v)$ to $\min\{d(w) + 1 : w \in \Gamma(v) \text{ and } rescap(v, w) > 0\}$. Despite these differences, our algorithm shares the essential properties of the generic algorithm of Goldberg and Tarjan. In particular, the proofs given in [GT88] of the next two lemmas and of Theorem 2.4 carry over without modification.

LEMMA 2.1 [GT88]. *For all* $v \in V, 0 \leq d(v) \leq 2n - 1$ *throughout the execution. In particular, the total number of relabeling operations is* $\leq n(2n - 1) \leq 2n^2$.

LEMMA 2.2 [GT88]. *The total number of saturating pushes is* $O(nm)$.

Our analysis uses the notion of an *undirected edge* $\{v, w\}$, i.e., a pair of directed edges (v, w) and (w, v). Define the capacity of an undirected edge $\{v, w\}$ to be $ucap(v, w) = cap(v, w) + cap(w, v)$. For all $v \in V$, let $\deg(v)$ denote the number of undirected edges incident with v.

Each vertex $v \in V$ has an *adjacency list*, which consists of all edges $(v, w) \in E$. For each $v \in V$, the first eligible edge (if any) in v's adjacency list is called its *current edge* and is denoted by $ce(v)$ ($ce(v) = nil$, if there are no eligible edges (v, w)). We study implementations of the generic maximum-flow algorithm that maintain, for each vertex v, a pointer to $ce(v)$. Lemma 2.3, whose proof is similar to those of Lemma 4.1 and Theorem 4.2 in [GT88], shows that this contributes $O(nm)$ time to the total running time.

LEMMA 2.3. *Maintaining current edges for all vertices over the whole execution can be done in* $O(nm)$ *time.*

Proof. Maintain for each vertex $v \in V$ a pointer z_v into the adjacency list of v. At each relabeling of v, initialize z_v to point to $ce(v)$. Whenever the edge pointed to by z_v becomes ineligible, step z_v through v's adjacency list, starting from its previous position, until it either points to an eligible edge or reaches the end of the list. The fact that z_v always points to $ce(v)$, or to the end of v's adjacency list if $ce(v) = nil$, follows from the observation that once an

edge (v, w) becomes ineligible, it remains so until the next relabeling of v. For each relabeling of v, the total time spent in maintaining z_v is $O(\deg(v))$. Summing over all relabelings of all vertices gives a total time of $O(\sum_{v \in V}(2n-1)\deg(v)) = O(nm)$. □

THEOREM 2.4 [GT88]. *Suppose that the algorithm terminates. Then, at termination, the preflow f is a maximum flow.*

3. The excess scaling algorithm. In this section we consider networks with integer capacities. We give a variant of the excess scaling algorithm of Ahuja and Orlin [AO89] and show that although this algorithm is not strongly polynomial, the number of pushes executed by it is $O(n^2 m)$. In the next section, we make appropriate modifications to the algorithm to obtain a strongly polynomial algorithm.

The excess scaling algorithm that we are about to present is an instance of the generic algorithm of §2, and except for the minor differences noted there, it is also an instance of the generic maximum-flow algorithm of Goldberg and Tarjan [GT88]. The implementation of the generic algorithm in [GT88, p. 929] repeatedly selects an active vertex and applies a *push/relabel* step to it. Rather than selecting an arbitrary active vertex, it turns out to be advantageous to select an active vertex with relatively large flow excess. The excess scaling algorithm maintains a parameter Δ; initially $\Delta = 2^{\lfloor \log U \rfloor}$, and $\Delta = 0$ at termination. The execution of the algorithm is partitioned into *phases* such that Δ stays fixed during each phase and decreases by a factor of 2 between consecutive phases. We use "Δ" to denote both this parameter and its value at some step of the execution; the context will resolve any ambiguity. By a Δ-*phase*, where $\Delta \in \mathbb{R}$, we mean a phase in which the value of the parameter equals Δ.

Consider a particular phase. At the start of the phase the algorithm satisfies the invariant that $e(v) \leq 2\Delta$, for all $v \in V - \{s, t\}$. The algorithm selects an active vertex v only if $e(v) \geq \Delta$. Furthermore, among all such vertices it selects one with minimum $d(v)$. Another invariant for the Δ-phase is that every push has value $\leq 2\Delta$. It follows that for all $v \in V - \{s, t\}$, $e(v) < 3\Delta$ always (cf. Fact 3.4).

To satisfy the 2Δ bound on the value of every push, the value of every push out of a vertex $v \in V - \{s, t\}$ is computed using a modification $\tilde{e}(v)$ of $e(v)$ defined as follows:

$$\text{If } e(v) \leq 2\Delta, \text{ then } \tilde{e}(v) = e(v), \text{ otherwise } \tilde{e}(v) = \Delta.$$

This achieves two things: First, the amount of flow sent is always $\leq 2\Delta$, as claimed above. Second, if at some point $e(v) > 2\Delta$, then the algorithm attempts to make $e(v)$ equal to zero during the same phase by selecting v twice. (This would not be achieved, for example, by letting $\tilde{e}(v) = \min\{e(v), 2\Delta\}$.)

procedure $\tilde{e}(v : vertex) : real$;
  return (**if** $e(v) \leq 2\Delta$ **then** $e(v)$ **else** Δ);
procedure *select*: *vertex*;
  precondition: $\exists v \in V - \{s, t\} : e(v) \geq \Delta$.
    Return any vertex $v \in V - \{s, t\}$ with $e(v) \geq \Delta$ and
      $d(v) = \min\{d(u) : u \in V - \{s, t\} \text{ and } e(u) \geq \Delta\}$;

The excess scaling algorithm:
  generic initialize;
  $\Delta := 2^{\lfloor \log U \rfloor}$;
  while $\Delta \geq 1$ **do**
  begin
    while $\exists v \in V - \{s, t\} : e(v) \geq \Delta$ **do**
    begin
      $v :=$ *select*;

```
        while ce(v) ≠ nil and rescap(ce(v)) ≤ ẽ(v) do
            push (ce(v), rescap(ce(v)));    (* a saturating push *)
        if ce(v) = nil and ẽ(v) > 0 then
            relabel(v)
        else
            if e(v) ≥ Δ then
                push (ce(v), ẽ(v));         (* a nonsaturating push *)
    end;
    Δ := ⌊Δ/2⌋;
end.
```

To prove the correctness of the algorithm and to analyze its running time, we need the following facts. These facts are also used in subsequent sections.

Fact 3.1. For every vertex $w \in V - \{s, t\}$, $e(w)$ does not increase while $e(w) \geq \Delta$.

Proof. Any vertex v selected while $e(w) \geq \Delta$ has $d(v) \leq d(w)$, by definition. Since pushes are executed out of selected vertices only, it follows that the head vertex u of each push has $d(u) < d(w)$. In other words, no flow is sent into w while $e(w) \geq \Delta$. □

Fact 3.2. The value of every push is $\leq 2\Delta$.

Proof. This follows from the definition of $\tilde{e}$. □

Fact 3.3. The value of every nonsaturating push is $\geq \Delta$.

Fact 3.4. For every $v \in V - \{s, t\}$, $e(v) \leq 2\Delta$ at the start of each phase, $e(v) < \Delta$ at the end of each phase, and $e(v) < 3\Delta$ always.

Proof. Consider any $v \in V - \{s, t\}$. After initialization, $e(v) \leq 2\Delta$ clearly holds. At the termination of each phase, $e(v) < \Delta$, and hence $e(v) \leq 2\Delta$ at the start of the next phase. Facts 3.1 and 3.2 together imply that $e(v) < 3\Delta$ always. □

THEOREM 3.5 [AO89]. *The excess scaling algorithm is partially correct.*

Proof. The following claim shows that the termination condition for the generic algorithm is satisfied; the correctness then follows from Theorem 2.4.

Claim. When the last phase (with $\Delta = 1$) terminates, there are no active vertices.

To see this, use induction on the number of steps executed to show that the preflow f, and hence also the flow excess e, has integer values throughout. Therefore, when the last phase terminates, $e(v) = 0$ for all $v \in V - \{s, t\}$ (Fact 3.4). □

The next two results are not used later; however, they are of interest since the first one shows that the number of push steps can be bounded independently of U, while the second one uses this to give a new bound on the running time.

LEMMA 3.6. *The algorithm executes* $O(n^2 \min\{m, \log U\})$ *nonsaturating pushes.*

Proof. Ahuja and Orlin [AO89] gave an $O(n^2 \log U)$ bound on the number of nonsaturating pushes for their algorithm. Since the algorithmm here is a variant of the one in [AO89], a variant of their analysis applies here and gives the same $O(n^2 \log U)$ bound. We show the bound of $O(n^2 m)$ using a potential argument. First, for each $v \in V$, let

$$\phi(v) = \begin{cases} 0, & \text{if } e(v) = 0, \\ 1, & \text{if } 0 < e(v) \leq 2\Delta, \\ 2, & \text{if } e(v) > 2\Delta. \end{cases}$$

Intuitively, $\phi(v)$ is the number of times that v can be selected before $e(v)$ drops to zero (cf. the definition of $\tilde{e}$). Now take

$$\Phi = \sum_{v \in V - \{s,t\}} \phi(v) \cdot d(v).$$

A nonsaturating push over an edge (v, w) always lowers $\phi(v)$ by 1 and never increases $\phi(w)$ by more than 1 (Fact 3.2). Since $d(v) = d(w) + 1$, it follows that every nonsaturating push decreases Φ by at least 1. By Lemma 2.1, a saturating push increases Φ by at most $2n$, and a relabeling clearly increases Φ by at most 2. Furthermore, only saturating pushes and relabelings can increase Φ; in particular, note that going from one phase to the next leaves Φ unchanged. $\Phi = 0$ initially and $\Phi \geq 0$ always, so the total decrease in Φ (due to nonsaturating pushes) is bounded by the total increase in Φ (due to saturating pushes and relabelings). Since there are $O(nm)$ saturating pushes (Lemma 2.2) and $O(n^2)$ relabelings (Lemma 2.1), the number of nonsaturating pushes is $O(n \cdot nm + n^2) = O(n^2 m)$. □

THEOREM 3.7. *The excess scaling algorithm runs in time* $O(nm + n \log U + n^2 \min\{m, \log U\})$.

Proof. The initialization can be done in time $O(m)$. Each *relabel* or *push* step takes time $O(1)$. When the *select* procedure is implemented using appropriate data structures [AO89], [GTT90], it contributes an overall running time proportional to the number of pushes plus $O(n)$ per phase. The number of phases is $\lfloor \log U \rfloor + 1$. The theorem now follows from Lemmas 2.1–2.3 and 3.6. □

The next fact is very useful and has a straightforward proof. In the context of the minimum-cost flow problem, analogous results are crucial for designing and analyzing efficient strongly polynomial algorithms (cf. Lemma 4 of [F86] and Lemma 6 of [O88]). The proof technique used here resembles that of Lemma 6 in [O88].

Fact 3.8. For every fixed Δ-phase and every $(v, w) \in E$, the increase of $f(v, w)$ during the phase and all subsequent phases is at most $20n^2\Delta$.

Proof. Focus first on the Δ-phase and consider the potential

$$\Phi = \sum_{u \in V - \{s,t\}} e(u) \cdot d(u).$$

Whenever $f(v, w)$ increases by c (due to a push over (v, w) with value c), Φ decreases by at least c. The total increase of $f(v, w)$ during the phase is therefore bounded by the total decrease of Φ. $\Phi \geq 0$ always, and at the start of the phase $\Phi \leq 4n^2\Delta$ (by Lemma 2.1 and Fact 3.4). No *push* operation increases Φ, and a *relabel* operation increases Φ by at most 3Δ (by Fact 3.4). Consequently, the total increase of Φ is at most $6n^2\Delta$ (by Lemma 2.1). It follows that the total decrease of Φ is at most $10n^2\Delta$. Therefore, the total increase of $f(v, w)$ during the phase is at most $10n^2\Delta$.

The result now follows by summing over the remaining phases and using the fact that Δ decreases by a factor of 2 between consecutive phases. □

We call (v, w) a *forward huge edge* if $rescap(v, w) > 20n^2\Delta$. By Fact 3.8, a forward huge edge will never again be saturated. An undirected edge $\{v, w\}$ is called *huge* if $ucap(v, w) > 40n^2\Delta$. If $\{v, w\}$ is huge, clearly one of the directed edges (v, w) or (w, v), say, (v, w), is a forward huge edge. If the reverse edge (w, v) is not also a forward huge edge, we call it a *reverse huge edge*. Notice that an (undirected) huge edge continues to be huge until the end of the execution. We denote the total number of saturating pushes over (reverse) huge edges by $\sharp$*huge sat pushes*.

4. The strongly polynomial algorithm. The excess scaling algorithm has two bottlenecks that make it inefficient in comparison with previously known fast strongly polynomial algorithms. The first bottleneck is the large amount of time spent in executing nonsaturating pushes. The standard way of avoiding this bottleneck is to store for each vertex v the current edge $ce(v)$ together with $rescap(ce(v))$ in a suitable data structure that supports the operation of sending flow over a path of stored current edges. Such data structures make it possible to send flow over a path of l current edges in a running time that is substantially less than

proportional to l. The most efficient data structure known for this purpose is the *dynamic trees* data structure of Sleator and Tarjan [ST83].

From the point of view of achieving a running time of $O(nm)$, this data structure has one drawback. When used in the straightforward way, it contributes a running time of $O(\log n)$ per saturating push, i.e., of $O(nm \log n)$ overall, since the number of saturating pushes is $O(nm)$. In the context of the generic algorithm of Goldberg and Tarjan, a more efficient way of using the data structure is to insert an edge only if at least one nonsaturating push will be executed over the edge. In other words, the current edge of a vertex v is inserted into the data structure only when a push is to be applied to v while $rescap(ce(v)) > \tilde{e}(v)$, i.e., if the push operation will not saturate v's current edge.

We do not know whether this simple heuristic alone decreases the number of dynamic trees operations sufficiently. However, we obtain a remarkable decrease by combining it with another heuristic:

> At the start of the execution, randomly permute
> the adjacency list of each vertex.

The effectiveness of this heuristic comes from its interaction with scaling. As the execution progresses and the parameter Δ decreases, many edges eventually become huge. Roughly speaking, in a situation where there are many forward huge edges, the sum over all vertices v of the number of times v changes its current edge is significantly less than nm, unless the adjacency lists are maliciously ordered. A quantitative analysis of the efficiency of randomly permuting the adjacency lists needs the notion of PTR events, and we present such an analysis in §6 after developing the necessary machinery.

The second bottleneck of the excess scaling algorithm is that there are $\lfloor \log U \rfloor + 1$ phases, each of which incurs a running time overhead of $O(n)$ for initializing data structures. This bottleneck is easy to avoid. Rather than using the standard scaling method of decreasing the parameter Δ by a factor of 2 between phases, we use "tight scaling" and decrease Δ as much as possible between phases, i.e., at the end of each phase Δ is set to the minimum of $\Delta/2$ and $\max\{e(v) : v \in V - \{s, t\}\}$. We use a heap data structure to store all the vertices $v \in V - \{s, t\}$, with the key $e(v)$ used to maintain the heap order. Tight scaling with a running time overhead of $O(\log n)$ per phase is easy to implement in this way.

A *heap* is a data structure that maintains a set of *items*, each with a real-valued *key*, under the following operations [FT87]:

make heap:	Return a new empty heap.
is empty(h):	If the heap h has no items, then return *true*; otherwise return *false*.
insert(i, h):	Insert a new item i with predefined key into the heap h.
find max(h):	Return an item of maximum key in the heap h.
delete max(h):	Delete an item of maximum key from the heap h and return it.
delete(i, h):	Delete the item i from the heap h.
increase key(c, i, h):	Increase the key of the item i in the heap h by adding the nonnegative real number c.

The two last operations assume that the position of i in h is known. The *Fibonacci heaps* data structure [FT87] supports a sequence of k *delete* or *delete max* operations and l other heap operations, starting with no heaps, in time $O(l + k \log l)$.

The actual implementation of the strongly polynomial algorithm uses two heaps, rather than one, in order to execute the *select* step efficiently. The *d_heap* contains all vertices $v \in V - \{s, t\}$ with $e(v) \geq \Delta$. The heap order is maintained according to the key $-d(v)$. The *d_heap* is needed for efficiently selecting a vertex v with minimum $d(v)$ among those

with $e(v) \geq \Delta$. The *e_heap* is a Fibonacci heap containing all vertices $v \in V - \{s, t\}$ with $e(v) < \Delta$. The heap order is maintained according to the key $e(v)$. The *e_heap* is needed for efficient updating of Δ. A Fibonacci heap is used because only $O(1)$ amortized time can be allowed per *increase key* operation.

The *dynamic trees* data structure of Sleator and Tarjan [ST83], [ST85], [T83] maintains a set of vertex-disjoint rooted trees in which each tree edge has an associated real value and is considered to be directed toward the root, i.e., from child to parent. We shall need the following dynamic trees operations:

find root(v):	Find and return the root of the tree containing the vertex v.
find value(v):	Find and return the value of the edge from the vertex v to its parent. This operation assumes that v is not a tree root.
find min(v):	Return the nonroot ancestor w of v with minimum *find value*(w). In the case of ties, the ancestor furthest from v is returned. This operation assumes that v is not a tree root.
add value(v, c):	Add the real number c to the value of every edge on the path from the vertex v to *find root*(v).
link(v, w, c):	Combine the trees containing the vertices v and w by making w the parent of v and giving the value c to the new edge from v to w. This operation assumes that v and w are in different trees and that v is a tree root.
cut(v):	Break the tree containing the vertex v into two trees by deleting the edge from v to its parent. This operation assumes that v is not a tree root.

A sequence of k dynamic trees operations, starting with a collection of n single-vertex trees, can be executed in $O(k \log n)$ time.

We use the dynamic trees data structure to maintain a spanning forest F of G that contains a subset of the current edges, where the value associated with an edge in F is its residual capacity. The algorithm represents the preflow f in one of two different ways: For $(v, w) \in E$, if $(v, w) \notin F$ and $(w, v) \notin F$, then $f(v, w)$ is stored explicitly (using an array $f : E \to \mathbb{R}$). Otherwise, if $(v, w) \in F$, then $f(v, w)$ is given implicitly by $cap(v, w) - rescap(v, w)$ and $f(w, v)$ is given by $-f(v, w)$. Whenever a tree edge (v, w) is cut, we must find its associated value (i.e., $rescap(v, w)$) and then update the current values of $f(v, w)$ and $f(w, v)$. Also, when the algorithm terminates, $f(v, w)$ and $f(w, v)$ must be computed for all $(v, w) \in F$. The procedures *Link* and *Cut* given below execute a *link* and a *cut* and incorporate these conventions for representing f.

procedure *Link*$(v : vertex)$;
  (* Insert the edge $ce(v)$ into F *)
    Let $w = head(ce(v))$;
    link$(v, w,$ *rescap*$(v, w))$;

procedure *Cut*$(v : vertex)$;
  (* Cut the tree edge $ce(v) \in F$ and restore f values *)
    Let $w =$ *head*$(ce(v))$;
    $f(v, w) :=$ *cap*$(v, w) -$ *find value*(v); $f(w, v) := -f(v, w)$;
    cut(v).

The heart of the algorithm is the procedure *macropush*. The algorithm repeatedly selects a vertex $v \in V - \{s, t\}$ with $e(v) \geq \Delta$ that has minimum label $d(v)$ among the vertices with flow excess $\geq \Delta$, after which the *macropush* procedure is applied to v.

If v is a nonroot vertex in F, i.e., $v \neq$ *find root*(v), then *macropush* uses the dynamic trees data structure to send flow from v, over a path of current edges, to *find root*(v). If one or

more edges become saturated, then all saturated edges in F are deleted using *cut* operations. The algorithm may execute more than one nonsaturating push while sending flow over a path in F.

Now suppose that v is a root in F. Then *macropush* executes zero or more saturating pushes over edges emanating from v, without inserting these edges into the dynamic trees data structure. After this, provided that the remaining flow excess of v is at least $\Delta/2$ and that there is an eligible edge emanating from v, *macropush* executes a sequence consisting of a *Link*(v) operation and a nonsaturating push over $ce(v)$. The reason for executing a *Link*(v) only if $e(v) \geq \Delta/2$ is to ensure that only edges with sufficiently large capacities relative to Δ are ever inserted into the dynamic trees data structure (cf. Fact 5.2).

In the following outline of the algorithm, the operations on heaps are not mentioned explicitly. The procedure $\tilde{e}(v)$ is repeated from the previous section, and the procedures *relabel* and *select* are more elaborate versions of identically named procedures in previous sections.

```
procedure  ẽ(v : vertex) : real;
    return (if e(v) ≤ 2Δ then e(v) else Δ);

procedure  relabel(v : vertex);
    for all u ∈ V with ce(u) = (u, v) do
        if (u, v) ∈ F then Cut(u);
    d(v) := d(v) + 1;

procedure  select: vertex;
    (* Return an active vertex v with e(v) ≥ Δ and minimum d(v) among the vertices having
    flow excess ≥ Δ, or decrease Δ and then select v as before, or return with Δ = 0 *)
    if ∀v ∈ V − {s, t} : e(v) < Δ then
        Δ := min{Δ/2, max{e(v) : v ∈ V − {s, t}}};
    if Δ > 0 then
        let v be any active vertex with e(v) ≥ Δ and
            d(v) = min{d(u) : u ∈ V − {s, t} and e(u) ≥ Δ}
    else
        let v := t;    (* dummy value *)
    return (v);

procedure  macropush(v : vertex);
    (* Send flow from v until v is relabeled or until e(v) decreases to < Δ/2 due to saturating
    pushes out of v or until flow is sent from v over a path in F *)
    if v = find root(v) then
        begin
        while ce(v) ≠ nil and rescap(ce(v)) ≤ ẽ(v) do
            push(ce(v), rescap(ce(v)));    (* a saturating push *)
        if ce(v) = nil and ẽ(v) > 0 then
            begin relabel(v); return; end
        else
            if ẽ(v) < Δ/2 then return
            else
                Link(v);    (* insert ce(v) with rescap(ce(v)) > ẽ(v) ≥ Δ/2 *)
        end;
```

(* Send as much flow as possible from v to *find root*(v), and then cut the tree edges in F that get saturated *)
$c := \min\{$*find value*(*find min*$(v))$, $\tilde{e}(v)\}$;
change value$(v, -c)$;
$e(v) := e(v) - c$; $e($*find root*$(v)) := e($*find root*$(v)) + c$;
while $v \neq$ *find root*(v) and *find value*(*find min*$(v)) = 0$ **do** *Cut*(*find min*$(v))$;

The strongly polynomial algorithm:

generic initialize;
initialize the *d_heap*, the *e_heap* and the dynamic trees data structure;
for each $v \in V$, randomly permute the adjacency list of v;
$\Delta := \infty$;

loop
  $v :=$ *select*;
  if $\Delta = 0$ **then stop;** (* f is a maximum flow *)
  macropush(v);
forever.

THEOREM 4.1. *The strongly polynomial algorithm is partially correct.*

5. Preliminaries for the analysis. After presenting some elementary facts about the strongly polynomial algorithm, we give a lemma that identifies the critical component of its running time.

The algorithm is easily seen to satisfy Facts 3.1, 3.2, 3.4, and 3.8.

Fact 5.1. Between consecutive phases, Δ decreases by a factor of 2 or more.

Fact 5.2. When a *link* operation is executed on an edge (v, w), then *rescap*$(v, w) \geq \Delta/2$.

Fact 5.3. For all $v \in V$, any increase of $e(v)$ while $v \neq$ *find root*(v) is caused by saturating pushes into v.

Proof. Any flow sent into v by a nonsaturating push is sent further all the way to *find root*(v); hence a nonsaturating push causes $e(v)$ to increase only if $v =$ *find root*(v). □

Fact 5.4. For all $v \in V$, if an execution of *macropush*(v) starting with $v \neq$ *find root*(v) does not execute any *cut*, then either $e(v) = 0$ or $e(v) > \Delta$ when the procedure terminates.

Proof. Clearly the total amount of flow sent from v by the procedure is $c = \tilde{e}(v)$. The claim now follows from the definition of $\tilde{e}(v)$. □

By a *select step* we mean one iteration of the main loop (**loop** . . . **forever**) of the algorithm. A vertex v is said to be *processed* by a *select* step if the call of the *select* procedure in that iteration returns v. Let $\sharp$*selects* denote the total number of *select* steps over the whole execution, i.e., the number of iterations of the main loop of the algorithm.

LEMMA 5.5. *The running time of the strongly polynomial algorithm is* $O(nm + \sharp selects \cdot \log n)$.

Proof. The total time for operations on the *d_heap* is $O(\sharp selects \cdot \log n)$, because there are $O(\sharp selects)$ operations on the *d_heap*. Next, consider operations on the *e_heap* and observe that we need not insert the vertex processed by a *select* step into the *e_heap* (if it belongs there) until the end of the *select* step. A saturating push over an edge (v, w) therefore causes one *increase key* operation, and if $e(w)$ becomes $\geq \Delta$, then w must be deleted from the *e_heap* and inserted into the *d_heap*. The number of *e_heap* delete operations at most equals the number of *d_heap* insert operations, which is $O(\sharp selects)$. The number of *increase key* operations is bounded by $\sharp selects$ plus the number of saturating pushes, which is $O(nm)$. The total time needed for operations on the *e_heap* is therefore $O(nm + \sharp selects \cdot \log n)$.

Consider the remaining running time, excluding the time for heap operations. The initialization can be done in $O(m)$ time; in particular, note that random permutations can be computed in linear time (see, e.g., [S77]). Each iteration of the main loop runs in time $O(\log n)$, plus $O(\log n)$ times the number of *cut* operations executed, plus the time for executing saturating pushes over edges not in the dynamic trees data structure, plus the time for maintaining current edges. The total number of *cut* operations executed is $\leq \sharp selects$, because there is a distinct *link* operation corresponding to each *cut*, and the number of *link* operations is clearly $\leq \sharp selects$, since each iteration of the main loop executes at most one *link*. The overall time for maintaining the current edges is $O(nm)$, and the overall time for saturating pushes that are not associated with *cut* operations is $O(nm)$. Thus, the total running time is $O(nm + \sharp selects \cdot \log n)$. □

LEMMA 5.6. *$\sharp selects = O(nm)$, and the worst-case running time of the strongly polynomial algorithm is $O(nm \log n)$.*

Proof. It is easily seen that the total number of *cut* and *link* operations is $O(nm)$. It follows that the number of iterations of the main loop that execute either a *cut* or a *link* or a *relabel* or a saturating push is $O(nm + n^2) = O(nm)$.

To handle the remaining *select* steps, call each such step a *neat select* step and note that each neat *select* step moves flow from a nonroot in F to a root. Define Φ as the sum over all nonroot vertices v of $\phi(v)$, where ϕ is the function of the proof of Lemma 3.6, i.e.,

$$\phi(v) = \begin{cases} 0, & \text{if } e(v) = 0, \\ 1, & \text{if } 0 < e(v) \leq 2\Delta, \\ 2, & \text{if } e(v) > 2\Delta. \end{cases}$$

By the previous observation, each neat *select* step decreases Φ by at least 1. On the other hand, only saturating pushes and link operations can increase Φ, and each such operation increases Φ by at most 2. Since $\Phi = 0$ initially, the number of neat *select* steps is $O(nm)$, and the same bound applies to the total number of *select* steps.

The bound on the running time now follows from the previous lemma. □

The crucial part of the analysis is to show that for sufficiently dense graphs, randomly permuting the adjacency lists causes $\sharp selects$ to become significantly less than $\Theta(nm)$ with high probability. In the next section we analyze the efficiency of randomly permuting the adjacency lists, and based on this in §7 we give a simple analysis of $\sharp selects$.

6. PTR events. A *premature target relabeling* event (PTR event) is defined to be the relabeling of the head w of a current edge (v, w). In other words, a PTR event may be identified with a triple (v, w, k), where $0 \leq k \leq 2n - 2$ and $(v, w) \in E$ is the current edge of $v \in V$ when the vertex w, which currently has $d(w) = k$, gets relabeled. By definition, every *cut* executed by the *relabel* procedure corresponds to a PTR event; however, there may be other PTR events besides these, since the dynamic trees data structure may not contain all current edges. Denote by $\sharp$ptr the total number of PTR events.

The significance of PTR events is that a vertex changes its current edge if and only if either a saturating push occurs over the edge or a PTR event occurs on the edge. Since no forward huge edge is ever saturated (see §3), a vertex whose current edge is a forward huge edge changes its current edge exactly when a PTR event occurs on the edge.

LEMMA 6.1. *Over the whole execution, $\sharp huge\ sat\ pushes \leq m + \sharp$ptr.*

Proof. Between two consecutive saturating pushes over a reverse huge edge (w, v), there must be a push over the forward huge edge (v, w). When the push over (v, w) is executed, then $(v, w) \in F$. Consequently, between this step and the next saturating push over (w, v), a *cut* on (v, w) must be executed. Hence $\sharp huge\ sat\ pushes$ is at most m plus the number of cuts on forward huge edges, which is $\leq m + \sharp$ptr. □

It is easy to bound ♯ptr by $O(nm)$. However, for sufficiently dense graphs a much tighter bound can be obtained by making use of the fact that each vertex randomly and independently permutes its adjacency list at the start of the execution. Before delving into the analysis, we introduce some notation whose usefulness will become evident below.

For every finite set A, let $\mathrm{Perm}(A)$ be the set of all permutations of A, i.e., of all bijections from $\{1, \ldots, |A|\}$ to A. Given finite sets A and B and permutations $\mu \in \mathrm{Perm}(A)$ and $\sigma \in \mathrm{Perm}(B)$, let $\lambda(\mu, \sigma)$, called the *coascent* of μ and σ, be the length of a longest (not necessarily contiguous) common subsequence of the sequences $\mu(1), \ldots, \mu(|A|)$ and $\sigma(1), \ldots, \sigma(|B|)$. Given l permutations $\mu_1, \ldots, \mu_l$ of subsets of a finite set A, let $\Lambda(\mu_1, \ldots, \mu_l) = \max_{\sigma \in \mathrm{Perm}(A)} \sum_{i=1}^{l} \lambda(\mu_i, \sigma)$; note that this quantity does not depend on A. We call $\Lambda(\mu_1, \ldots, \mu_l)$ the *external coascent* of $\mu_1, \ldots, \mu_l$.

Example. This example is meant to familiarize the reader with the definitions of λ and Λ. For the duration of the example, we identify a permutation μ of a set B with the string $\mu(1) \ldots \mu(|B|)$ and use as our universe the set $A = \{a, b, c, d, e, f\}$ of six symbols.

Let $\mu = bead$ and $\sigma = fbadec$. Then $\lambda(\mu, \sigma) = 3$, since the sequence *bad* occurs in both μ and σ, whereas the only longer subsequence *bead* occurring in μ does not occur in σ.

Now take $\mu_1 = fade$, $\mu_2 = bead$ and $\mu_3 = dec$. Then $\Lambda(\mu_1, \mu_2, \mu_3) = 10$, since with $\sigma = fbadec$ we have $\sum_{i=1}^{3} \lambda(\mu_i, \sigma) = 10$, whereas it is not difficult to see that there is no permutation σ' of A with $\sum_{i=1}^{3} \lambda(\mu_i, \sigma') = 11$.

Identify V with the set $\{1, \ldots, n\}$ and let $\mu_v \in \mathrm{Perm}(\Gamma(v))$, for all $v \in V$. We shall say that the strongly polynomial algorithm is executed with the adjacency lists ordered according to $\mu_1, \ldots, \mu_n$ if the following holds after the initialization: For all $u \in V$ and all $v, w \in \Gamma(u)$, the edge (u, v) precedes (u, w) in u's adjacency list if and only if $\mu_u^{-1}(v) < \mu_u^{-1}(w)$. Furthermore, for $\sigma_0, \ldots, \sigma_{2n-2} \in \mathrm{Perm}(V)$, let us say that an execution of the algorithm relabels according to $\sigma_0, \ldots, \sigma_{2n-2}$ if the following holds for all k with $0 \le k \le 2n-2$ and all $v, w \in V$: If $d(v)$ is set to $k+1$ at some point of the execution and $d(w)$ is set to $k+1$ at some later point, then $\sigma_k^{-1}(v) < \sigma_k^{-1}(w)$. Except for the fact that some vertices may not be relabeled $k+1$ times, σ_k simply orders the vertices in V by the time of their $(k+1)$st relabeling.

Consider now an execution of the algorithm with the adjacency lists ordered according to $\mu_1, \ldots, \mu_n$ that relabels according to $\sigma_0, \ldots, \sigma_{2n-2}$. Fix $v \in V$ and k with $0 \le k \le 2n-2$ and suppose that for some vertices $w_1, \ldots, w_l \in V$, the execution incurs PTR events $(v, w_1, k), \ldots, (v, w_l, k)$, in that order. Then, clearly $\mu_v^{-1}(w_1) < \cdots < \mu_v^{-1}(w_l)$ and $\sigma_k^{-1}(w_1) < \cdots < \sigma_k^{-1}(w_l)$, i.e., the sequences $\mu_v(1), \ldots, \mu_v(|\Gamma(v)|)$ and $\sigma_k(1), \ldots, \sigma_k(n)$ have a (not necessarily contiguous) common subsequence of length l, namely $w_1, \ldots, w_l$. It follows that for all $v \in V$ and all k with $0 \le k \le 2n-2$, the total number of PTR events of the form (v, w, k), where $w \in V$, is bounded by $\lambda(\mu_v, \sigma_k)$. Summing over all $v \in V$ for fixed k with $0 \le k \le 2n-2$, we see that the total number of PTR events of the form (v, w, k), where $v, w \in V$, is bounded by

$$\sum_{v \in V} \lambda(\mu_v, \sigma_k) \le \Lambda(\mu_1, \ldots, \mu_n).$$

A final summation over all values of k yields the next lemma.

LEMMA 6.2. *For all $v \in V$, let $\mu_v \in \mathrm{Perm}(\Gamma(v))$. If the strongly polynomial algorithm is executed with the adjacency lists ordered according to $\mu_1, \ldots, \mu_n$, then* ♯ptr $\le 2n \cdot \Lambda(\mu_1, \ldots, \mu_n)$.

LEMMA 6.3. *Let A be a finite set with $|A| = N$, let $A_1, \ldots, A_N$ be subsets of A and take $M = \sum_{i=1}^{n} |A_i|$. Suppose that μ_i is drawn randomly from the uniform distribution over* $\mathrm{Perm}(A_i)$, *for $i = 1, \ldots, N$, and that $\mu_1, \ldots, \mu_N$ are independent. Then for all $r \ge \sqrt{NM} + N \log N$, $\Lambda(\mu_1, \ldots, \mu_N) = O(r)$ with probability at least $1 - 2^{-r}$.*

Proof. Recall that $\Lambda(\mu_1, \ldots, \mu_N) = \max_{\sigma \in \mathrm{Perm}(A)} \psi(\sigma)$, where $\psi(\sigma) = \sum_{i=1}^{n} \lambda(\mu_i, \sigma)$. We will show the probability that $\psi(\sigma)$ is large to be very small for each fixed $\sigma \in \mathrm{Perm}(A)$. Multiplying that probability by the number of choices for σ, i.e., by $N!$, we obtain an upper bound on the probability that $\Lambda(\mu_1, \ldots, \mu_N)$ is large.

Hence, let $\sigma \in \mathrm{Perm}(A)$ be arbitrary but fixed. For $i = 1, \ldots, N$, let $\Lambda_i = \lambda(\mu_i, \sigma)$ and take $S = \psi(\sigma) = \sum_{i=1}^{n} \Lambda_i$, the quantity of interest. For $i = 1, \ldots, N$, denote $|A_i|$ by a_i. Assume $N \geq 2$.

For arbitrary integers a and k with $0 \leq k \leq a \leq N$, the number of permutations μ of an arbitrary subset of A of cardinality a with $\lambda(\mu, \sigma) \geq k$ is at most $\binom{a}{k}^2 (a-k)!$. To see this, note that if $\lambda(\mu, \sigma) \geq k$, then the elements of a (not necessarily contiguous) subsequence of $\mu(1), \ldots, \mu(a)$ of length k appear in the same order in the sequence $\sigma(1), \ldots, \sigma(N)$. The elements of the subsequence can be chosen in $\binom{a}{k}$ ways, and the positions in which they appear in $\mu(1), \ldots, \mu(a)$ can also be chosen in $\binom{a}{k}$ ways, while the remainder of the sequence $\mu(1), \ldots, \mu(a)$ can be chosen in $(a-k)!$ ways. It follows that for $i = 1, \ldots, N$ and for all integers k with $1 \leq k \leq a_i$,

$$\Pr(\Lambda_i \geq k) \leq \frac{\binom{a_i}{k}^2 (a_i - k)!}{a_i!} \leq \frac{a_i^k}{(k!)^2} \leq \left(\frac{e^2 a_i}{k^2}\right)^k.$$

It can be seen that Λ_i is unlikely to exceed $\sqrt{a_i}$ by very much. By applying the Cauchy–Schwarz inequality $|u \cdot v| \leq |u|\,|v|$ to the vectors $u = (1, \ldots, 1)$ and $v = (\sqrt{a_1}, \ldots, \sqrt{a_N})$, we obtain

$$\sum_{i=1}^{N} \sqrt{a_i} \leq \sqrt{N} \cdot \sqrt{\sum_{i=1}^{N} a_i} = \sqrt{NM}.$$

Hence $S = \sum_{i=1}^{N} \Lambda_i$ is unlikely to exceed $\sqrt{NM}$ by very much. We now establish a precise bound.

First observe that for arbitrary $x \in \mathbb{R}$,

$$\Pr(S \geq x) = e^{-x} e^{x} \Pr(e^S \geq e^x) \leq e^{-x} E(e^S).$$

Second, since $\mu_1, \ldots, \mu_N$ and hence also $e^{\Lambda_1}, \ldots, e^{\Lambda_N}$ are independent,

$$E(e^S) = E\left(e^{\sum_{i=1}^{N} \Lambda_i}\right) = E\left(\prod_{i=1}^{N} e^{\Lambda_i}\right) = \prod_{i=1}^{N} E(e^{\Lambda_i}).$$

We next bound the quantities $E(e^{\Lambda_i})$. Let $i \in \{1, \ldots, N\}$ and let $b_i \geq 0$ be an arbitrary integer. Then

$$E(e^{\Lambda_i}) = \sum_{k=0}^{\infty} e^k \Pr(\Lambda_i = k) \leq \sum_{k=0}^{b_i} e^k \Pr(\Lambda_i = k) + \sum_{k=b_i+1}^{\infty} e^k \Pr(\Lambda_i \geq k)$$

$$\leq e^{b_i} \sum_{k=0}^{b_i} \Pr(\Lambda_i = k) + \sum_{k=b_i+1}^{\infty} e^k \left(\frac{e^2 a_i}{k^2}\right)^k \leq e^{b_i} + \sum_{k=b_i+1}^{\infty} \left(\frac{e^3 a_i}{k^2}\right)^k.$$

Choose b_i to make $\frac{e^3 a_i}{k^2} \leq \frac{1}{2}$ for $k \geq b_i + 1$, i.e., take $b_i = \lfloor \sqrt{2e^3 a_i} \rfloor$. Then

$$E(e^{\Lambda_i}) \leq e^{b_i} + \sum_{k=b_i+1}^{\infty} 2^{-k} = e^{b_i} + 2^{-b_i} \leq 2e^{\sqrt{2e^3 a_i}}.$$

Putting together everything yields

$$\Pr(S \geq x) \leq e^{-x} E(e^S) = e^{-x} \prod_{i=1}^{N} E(e^{\Lambda_i}) \leq e^{-x} \prod_{i=1}^{N} (2e^{\sqrt{2e^3 a_i}})$$

$$= e^{-x} \cdot 2^N e^{\sqrt{2e^3} \sum_{i=1}^{N} \sqrt{a_i}} \leq 2^N e^{\sqrt{2e^3}\sqrt{NM} - x}.$$

Recalling that σ can be chosen in $N!$ ways, we find

$$\Pr(\Lambda(\mu_1, \ldots, \mu_N) \geq x) \leq N! \cdot 2^N e^{\sqrt{2e^3}\sqrt{NM} - x}$$

$$\leq e^{2N \log N + \sqrt{2e^3}\sqrt{NM} - x}.$$

Given any $r \geq \sqrt{NM} + N \log N$, now choose

$$x = 2N \log N + \sqrt{2e^3}\sqrt{NM} + r = O(r)$$

and observe that

$$\Pr(\Lambda(\mu_1, \ldots, \mu_N) \geq x) \leq 2^{-r}. \qquad \square$$

LEMMA 6.4. *For every* $\alpha \geq 1$, $\sharp\text{ptr} = O(\alpha \cdot (n^{3/2} m^{1/2} + n^2 \log n))$ *with probability at least* $1 - 2^{-\alpha(\sqrt{nm} + n \log n)}$.

Proof. Combine Lemmas 6.2 and 6.3 $\square$

7. A simple analysis of operations on nonhuge edges and of the overall running time. In order to complete the analysis of the running time, i.e., to bound $\sharp$*selects*, we have to show good bounds for the total number of operations on nonhuge edges. In this section we derive somewhat loose bounds by using a simple argument, and in the next section we give better bounds by using a more refined argument.

Let β be a real number with $2 \leq \beta = n^{O(1)}$ (we fix $\beta = 2 + \sqrt{m/n}$ to get the main result). Define the *status* of an undirected edge $\{v, w\}$ as follows:

$\{v, w\}$ is said to be *small* if $ucap(v, w) < \Delta/\beta$, *medium* if $\Delta/\beta \leq ucap(v, w) \leq 40n^2\Delta$, and (as defined in §3) *huge* if $ucap(v, w) > 40n^2\Delta$. Note that the status of an edge may change during the execution, but at most twice.

A push is called *small*, *medium*, or *huge*, respectively, if it is executed over a small edge, a medium edge, or a huge edge. We denote the number of medium saturating pushes by $\sharp$*med sat pushes*.

There is an obvious bound of $O(nm)$ of $\sharp$*med sat pushes*. However, for sufficiently dense graphs $\sharp$*med sat pushes* can be shown to be significantly less than proportional to nm. Intuitively, the reason for the relatively small number of saturating pushes over medium edges is that an edge is medium for only $O(\log n)$ phases, i.e., medium edges are "short-lived."

In order to bound the number of medium saturating pushes, we partition these into three classes, (a), (b), and (c). To define the partition, consider a medium saturating push over an edge (v, w). If either $d(v)$ or $d(w)$ has the same value at the time of the push as at a "phase boundary," i.e., at the beginning or at the end of a phase, then the push is a class (a) push. Otherwise, if the number of medium (undirected) edges incident with w at the time of the push is $\leq \deg(w)/\beta$, it is a class (b) push. If neither of these cases applies, the push is a class (c) push. Note that if a push of class (b) or (c) is executed over an edge (v, w) during a particular period of eligibility of (v, w), then that period of eligibility of (v, w) lies entirely within one phase.

LEMMA 7.1. *Over the whole execution, the number of class* (a) *and class* (b) *medium saturating pushes is* $O(nm/\beta + m \log n)$.

Proof. The number of class (a) pushes is $O(m \log n)$ because each edge is medium for only $O(\log n)$ phases, and between two successive saturating pushes over an edge both its tail vertex and its head vertex have to be relabeled.

The number of class (b) pushes is $O(nm/\beta)$. To see this, note that between two consecutive relabelings of a vertex w there are $\leq \deg(w)/\beta$ medium saturating pushes of class (b) into w. Summing over all $w \in V$ and all relabelings of w gives $O(\sum_{w \in V}(2n - 1) \deg(w)/\beta) = O(nm/\beta)$. □

The medium saturating pushes of class (c) are more difficult to handle, and we need a few more results before we can tally them.

Define the *throughput* of a saturating push over an edge (v, w) while $d(v) = k$ to be equal to $rescap(v, w)$ when $d(v)$ is set to k. In other words, the throughput is the total amount of flow sent over (v, w) during a maximal period of eligibility of (v, w). Note that the value of the saturating push may be less than its throughput.

LEMMA 7.2. *Over the whole execution, the number of nonsmall saturating pushes with throughput* $< \Delta/\beta$ *is* $\leq m + \sharp ptr$.

Proof. Consider a nonsmall saturating push over an edge (v, w) while $d(v) = k$. Since $ucap(v, w) \geq \Delta/\beta$, it follows that when $d(v)$ is set to k, then $rescap(w, v) > 0$; also, obviously, $rescap(v, w) > 0$. Assume that either (v, w) or (w, v) has been used as a current edge before this use of (v, w) as a current edge. It can then be seen that the previous use of (v, w) or (w, v) as a current edge was terminated by a PTR event. The lemma follows. □

The medium saturating pushes of class (c) that have throughput $< \Delta/\beta$ are easily taken care of by the previous lemma, so we now turn our attention to the remaining class (c) pushes. Recall that all the pushes contributing to the throughput of a class (c) push are executed in the same phase.

Consider a vertex w that is the head vertex of a medium saturating push of class (c). By the definition of class (c), a fraction of more than $1/\beta$ of the undirected edges incident with w are medium. The next lemma enables us to focus on such vertices and to give a sufficiently good bound on the number of class (c) saturating pushes with throughput $\geq \Delta/\beta$ into these vertices. The lemma is a generalization of Lemma 6 of [AO89], whose proof, with a minor modification, shows that the number of pushes with value $\geq \Delta$ in any phase is $O(n^2)$.

LEMMA 7.3. *For every subset* V' *of* V *and in every fixed* Δ*-phase, the number of pushes with value* $\geq \Delta$ *into the vertices of* V' (*i.e., pushes over edges of the form* (v, w), *where* $w \in V'$) *is* $O(|V'| \cdot n + D)$, *where* D *denotes the number of relabel operations executed in the phase.*

Since our main objective at this point is to provide intuition, we prove the lemma only for the special case when all the edge capacities are integers; the proof trivially extends to the case of rational edge capacities. A direct but less intuitive proof of the general case of the lemma is given in the next section (Lemma 8.3).

Suppose that all the edge capacities are integers, and also assume that the algorithm uses integer division, i.e., the occurrence of $\Delta/2$ in the procedure *select* is replaced by $\lfloor \Delta/2 \rfloor$. (Note that the test $\tilde{e}(v) < \Delta/2$ in the procedure *macropush* can be carried out without division.) Induction on the number of steps executed shows the preflow f, and hence also the flow excess e, to have integer values throughout the execution, and it can be seen that this modification does not affect our analysis of the algorithm. Furthermore, each unit of flow remains an indivisible entity throughout the execution. Call each unit of flow a *flow atom*. Related notions of flow atoms were previously used to analyze maximum-flow algorithms by, for example, Shiloach and Vishkin [SV82], Goldberg [G85], Cheriyan and Maheshwari [CM89], and Tunçel [T90].

Proof. (Lemma 7.3, integer edge capacities). The lemma is a straightforward consequence of a property of the so-called moving sequence of a flow atom. Consider a flow atom q and define $d(q)$ to be $d(v)$, where v is the vertex at which q is currently located. Focus on the movement of q during the phase under consideration and note that each change of $d(q)$ is caused by either a push operation or a relabel operation applied to the vertex at which q is currently located.

The *moving sequence* of q, $ms(q)$, is a string over the alphabet $\{\langle u, i\rangle : u \in V' \text{ and } 0 \leq i \leq 2n - 1\} \cup \{\uparrow\}$ defined as follows: At the start of the phase, $ms(q)$ is empty. If q is pushed into a vertex $u \in V'$ (i.e., q is sent over an edge whose head vertex u is in V'), $\langle u, d(u)\rangle$ is appended to $ms(q)$, and if the vertex currently holding q undergoes a relabeling, then an $\uparrow$ is appended to $ms(q)$. Let $|ms(q)|_{\uparrow}$ denote the number of $\uparrow$ symbols in a moving sequence $ms(q)$, and let $|ms(q)|_{\downarrow}$ denote the number of remaining symbols in $ms(q)$, i.e., the number of symbols of the form $\langle u, i\rangle$. Finally let $n' = |V'|$.

The main property of moving sequences is

$$|ms(q)|_{\downarrow} \leq n' + |ms(q)|_{\uparrow}.$$

To show this relation, assume that $ms(q)$ contains at least one non-$\uparrow$ symbol, and let $\langle u, i\rangle$ be the first of these. Consider the prefix of $ms(q)$ up to and including the last symbol in $ms(q)$ with a first component of u and denote this prefix by $ms'(q)$. It is easy to show that $|ms'(q)|_{\downarrow} \leq 1 + |ms'(q)|_{\uparrow}$. To see this, note that each $\uparrow$ symbol corresponds to an increase of $d(q)$ by one, while each non-$\uparrow$ symbol corresponds to a decrease of $d(q)$ by one. Furthermore, by comparing the first non-$\uparrow$ symbol $\langle u, i\rangle$ in $ms'(q)$ with the symbol $\langle u, j\rangle$ at the end of $ms'(q)$ and noting that $d(u)$ is nondecreasing throughout the execution, we may conclude that $j \geq i$. Consequently, the number of non-$\uparrow$ symbols in $ms'(q)$ is at most one more than the number of $\uparrow$ symbols.

Now delete $ms'(q)$ from $ms(q)$. If the remaining string contains any non-$\uparrow$ symbols, let $\langle v, l\rangle$ be the first of these and consider the prefix $ms''(q)$ consisting of all symbols up to and including the last symbol with a first component of v. Using the same argument, it can be seen that $|ms''(q)|_{\downarrow} \leq 1 + |ms''(q)|_{\uparrow}$.

Repeating this argument a total of at most n' times shows that

$$|ms(q)|_{\downarrow} \leq n' + |ms(q)|_{\uparrow}.$$

Let Q denote the set of all flow atoms that are located at active vertices at the start of the phase (i.e., flow atoms located at s or t at the start of the phase are not in Q). Fact 3.4 clearly implies that $|Q| \leq 2n \cdot \Delta$. To count the number of pushes with value $\geq \Delta$ into the vertices of V', notice that each of these operations pushes $\geq \Delta$ flow atoms into some vertex $u \in V'$ and causes a symbol with a first component of u to be appended to the moving sequence of each of these flow atoms. Hence, the number of such pushes is

$$\begin{aligned}
&\leq (1/\Delta) \sum_{q \in Q} |ms(q)|_{\downarrow} \\
&\leq (1/\Delta) \sum_{q \in Q} (n' + |ms(q)|_{\uparrow}) \\
&\leq (1/\Delta)((2n \cdot \Delta \cdot n') + \sum_{q \in Q} |ms(q)|_{\uparrow}) \\
&\leq 2nn' + (1/\Delta)(3\Delta \cdot D) = 2nn' + 3D.
\end{aligned}$$

The last inequality follows because there are D *relabel* operations, and the total number of $\uparrow$ symbols appended by a single relabeling is at most 3Δ (Fact 3.4). □

We actually use a more technical but straightforward generalization of the previous lemma.

COROLLARY 7.4. *For every subset V' of V and in every fixed Δ-phase, the number of class* (c) *medium saturating pushes with throughput $\geq \Delta/\beta$ over edges of the form (v, w), where $w \in V'$, is $O(\beta(|V'| \cdot n + D))$, where D denotes the number of relabel operations executed in the phase.*

We need a technical definition. A Δ-phase is said to *hit* a vertex v if the number of medium (undirected) edges incident with v in the phase is $> \deg(v)/\beta$. Notice that each phase hits the head vertex of every class (c) medium saturating push executed in the phase. The following interpretation may be useful below: associate each undirected edge $\{v, w\}$ with the interval $[ucap(v, w)/(40n^2),\ \beta \cdot ucap(v, w)]$ on the x axis, and view "the phase" as a vertical line sweeping the x axis from ∞ to 0; the current value of Δ gives the current location of the sweep line, and an edge is medium exactly if its associated interval is currently intersected by the sweep line.

Fact 7.5. For all $v \in V$, the number of distinct phases that hit v is $O(\beta \log n)$. Therefore the sum over all phases of the number of vertices hit by the phase is $O(n\beta \log n)$.

Proof. Let the phases that hit v have parameters $\Delta_1, \Delta_2, \ldots$, and for $i = 1, 2, \ldots$, let $M_i(v)$ denote the set of medium (undirected) edges incident with v in the Δ_i-phase. The number of Δ_i- phases ($i \geq 2$) that hit v such that $M_1(v) \cap M_i(v) \neq \emptyset$ is $O(\log n)$, because in each such phase the parameter Δ_i is in the interval $[\Delta_1/(40n^2\beta), 40n^2\beta\Delta_1]$.

If v is hit by yet another phase (besides the $O(\log n)$ phases enumerated above) with parameter, say, $\Delta_j (j \geq 2)$, then $M_1(v) \cap M_j(v) = \emptyset$, and both $|M_1(v)|$ and $|M_j(v)|$ are $> \deg(v)/\beta$. Continuing the argument, it follows that v is hit by $O(\beta \log n)$ phases. □

LEMMA 7.6. *Over the whole execution, the number of class* (c) *medium saturating pushes is $O(n^2\beta^2 \log n + \sharp\text{ptr})$.*

Proof. By Lemma 7.2, the number of medium saturating pushes with throughput $< \Delta/\beta$ is $O(m + \sharp\text{ptr})$.

Consider a fixed Δ-phase and focus on the class (c) medium saturating pushes with throughput $\geq \Delta/\beta$ in that phase. Let V' denote the set of vertices hit by the phase, and let D denote the number of *relabel* operations in the phase. By applying Corollary 7.4, we can see that the number of class (c) medium saturating pushes with throughput $\geq \Delta/\beta$ is $O(\beta(|V'| \cdot n + D))$.

Finally, by summing over all phases and using Fact 7.5 and the fact that the total number of relabelings is $O(n^2)$, we find that the total number of class (c) medium saturating pushes with throughput $\geq \Delta/\beta$ is $O(n^2\beta^2 \log n)$. □

We are now ready to bound $\sharp$*selects* and thereby complete the simple analysis of the running time.

LEMMA 7.7. $\sharp selects = O(nm/\beta + n^2\beta^2 \log n + \sharp\text{ptr})$.

Proof. Every *cut* operation is associated with either a PTR event or a nonsmall saturating push, since no small edge is ever inserted into the dynamic trees data structure (Fact 5.2). Hence the number of *cut* operations is bounded by $\sharp$*med sat pushes* + $\sharp$*huge sat pushes* + $\sharp$ptr. The number of *link* operations clearly exceeds the number of *cut* operations by at most $n - 1$ because the forest F never contains more than $n - 1$ edges, and the number of *relabel* operations is $O(n^2)$. Therefore the number of *select* steps that execute either a *cut* or a *link* or a *relabel* is $O(n^2 + \sharp$*med sat pushes* + $\sharp$*huge sat pushes* + $\sharp$ptr).

The remaining *select* steps can be partitioned into two classes: those that execute one or more saturating pushes, and those that do not execute any saturating push.

The number of *select* steps that execute a saturating push and do not execute any *cut*, *link*, or *relabel* is $O(nm/\beta + \sharp$*med sat pushes* + $\sharp$*huge sat pushes*), because if a *select* step executes only small saturating pushes, then it executes at least $\beta/2$ saturating pushes, since

the flow excess of the processed vertex decreases from $\geq \Delta$ to $< \Delta/2$ and each small push has value $< \Delta/\beta$.

Now consider the *select* steps that do not execute any *cut*, *link*, *relabel*, or saturating push, and call each such step a *neat select* step. (The argument here is similar to that in the proof of Lemma 5.6.) The vertex processed by a neat *select* step is a nonroot in F throughout that step. Consider a fixed vertex v and focus on the part of the execution between two consecutive nonneat *select* steps that process v, or after the last such *select* step. Suppose that v is processed by one or more neat *select* steps in this part of the execution. When the first of these neat *select* steps is executed, then $\Delta \leq e(v) < 3\Delta$; hence it follows by Fact 5.4 that $e(v)$ decreases to zero after at most two neat *select* steps that process v, and over the whole execution, this gives a number of neat *select* steps that is at most twice the number of nonneat *select* steps. Before yet another neat *select* step that processes v, $e(v)$ has to increase from zero to $\geq \Delta$. Further, by Fact 5.3, any increase of $e(v)$ is caused by saturating pushes into v. Consider these saturating pushes and their associated edges (u, v), where $u \in V$. If each of these edges (u, v) has $ucap(u, v) < \Delta/\beta$ when v is processed by the neat *select* step, then clearly there are $\geq \beta$ saturating pushes into v, and over the whole execution this case gives $O(nm/\beta)$ neat *select* steps. Otherwise, either there is at least one nonsmall saturating push into v or the status of one of these edges changes (from small to medium or huge) between the earliest of these saturating pushes and the neat *select* step that processes v. Over the whole execution, the former case gives $O(\sharp \textit{med sat pushes} + \sharp \textit{huge sat pushes})$ neat *select* steps, and the latter case gives $O(m)$ neat *select* steps, because the total number of status changes of edges is $\leq 2m$.

Therefore, the total number of neat *select* steps is at most twice the total number of nonneat *select* steps, plus $O(m + nm/\beta + \sharp \textit{med sat pushes} + \sharp \textit{huge sat pushes})$. Hence

$$\begin{aligned} \sharp \textit{selects} &= O(nm/\beta + n^2 + \sharp \textit{med sat pushes} + \sharp \textit{huge sat pushes} + \sharp \text{ptr}) \\ &= O(nm/\beta + n^2\beta^2 \log n + \sharp \text{ptr}). \end{aligned}$$

For the second equation, we use the bound on $\sharp$*med sat pushes* given by Lemmas 7.1 and 7.6 and the bound on $\sharp$*huge sat pushes* given by Lemma 6.1. □

LEMMA 7.8. *For every $\alpha \geq 1$, the strongly polynomial maximum-flow algorithm runs in time $O(\alpha \cdot (nm + n^{4/3}m^{2/3}(\log n)^{4/3} + n^2(\log n)^2))$ with probability at least $1 - 2^{-\alpha(\sqrt{nm} + n \log n)}$.*

Proof. By taking $\beta = 2 + (m/(n \log n))^{1/3}$ and using the previous lemma and the bound on $\sharp$ptr given in Lemma 6.4, we see that $\sharp \textit{selects} = O(n^{4/3}m^{2/3}(\log n)^{1/3} + \alpha \cdot (n^{3/2}m^{1/2} + n^2 \log n))$ with probability at least $1 - 2^{-\alpha(\sqrt{nm} + n \log n)}$. The bound on the running time now follows from Lemma 5.5. □

For $m = \Omega(n(\log n)^4)$ the running time is $O(nm)$, with high probability.

8. A strengthened analysis of operations on medium edges and of the overall running time. An examination of the analysis in the previous section shows that the bottleneck in bounding $\sharp$*selects* is the contribution due to $\sharp$*med sat pushes*, since our bound for $\sharp$*med sat pushes* is $O(n^{4/3}m^{2/3}(\log n)^{1/3} + \sharp \text{ptr})$, whereas the bound for both $\sharp$*huge sat pushes* and $\sharp$ptr is $O(n^{3/2}m^{1/2} + n^2 \log n)$. In this section, a tighter analysis of the contribution of $\sharp$*med sat pushes* to $\sharp$*selects* is developed.

Recall that the medium saturating pushes are partitioned into three classes, (a), (b), and (c), and that the number of pushes in these classes are $O(m \log n)$, $O(nm/\beta)$, and $O(n^2\beta^2 \log n + \sharp \text{ptr})$, respectively, for any β with $2 \leq \beta = n^{O(1)}$. The bottleneck term in the bound for $\sharp$*med sat pushes* is due to the class (c) pushes.

In order to bound the contribution of the class (c) pushes to $\sharp$*selects*, we partition this class into two subclasses. A push over an edge (v, w) is called *terminal* if at the time of the push

$$|\{u : u \in V' \text{ and } d(u) = d(w)\}| < \beta,$$

where V' denotes the set of vertices hit by the current Δ-phase, i.e., if at the time of the push there are fewer than β "hit vertices" with labels equal to $d(w)$. Otherwise the push is *nonterminal*.

LEMMA 8.1. *Over the whole execution, the number of terminal pushes of class* (c) *is* $O(n^2\beta)$.

Proof. Fix $v \in V$ and $k \in \{1, \ldots, 2n-1\}$ and recall that if there are any class (c) pushes out of v at all while $d(v) = k$, then all pushes out of v while $d(v) = k$ occur in the same phase, i.e., the set V' does not change between the first and the last of these pushes. Since an edge out of v can become eligible only during a relabeling of v, while at the time of a terminal push out of v there are fewer than β eligible edges of the form (v, w), where $w \in V'$, it is now clear that the number of terminal class (c) pushes out of v while $d(v) = k$ is bounded by β. Summing over all $v \in V$ and all $k \in \{1, \ldots, 2n-1\}$ yields a total of $O(n^2\beta)$ terminal class (c) pushes. □

The next lemma is the key instrument for bounding the contribution of the nonterminal class (c) pushes to $\sharp$*selects*.

LEMMA 8.2. *The sum over all Δ-phases of the ratio of the total value of all nonterminal pushes in the Δ-phase to $\Delta/2$ is $O(n^2 \log n)$.*

To see the relevance of this lemma, notice that it directly gives an $O(n^2 \log n)$ bound on the number of nonterminal pushes with value $\geq \Delta/2$. To prove Lemma 8.2, we need a stronger version of Lemma 7.3. The following lemma may be used to bound the total value of nonterminal pushes in any Δ-phase by taking V' (in the lemma) to be the set of vertices hit by the phase and taking γ (in the lemma) to be equal to β.

LEMMA 8.3. *Let γ be a number ≥ 1. For every subset V' of V and in every fixed Δ-phase, the total value of pushes into vertices w such that when the push is executed*

$$|\{u : u \in V' \text{ and } d(u) = d(w)\}| \geq \gamma$$

is $O((|V'| \cdot n/\gamma + D)\Delta)$, where D denotes the number of relabel operations executed in the phase.

Proof. Let $h = |V'|$ and $V' = \{v_1, \ldots, v_h\}$ and for all $v \in V$, define the *fooling height* of v as

$$d'(v) = \max_{i_1 \geq d(v_1), \ldots, i_h \geq d(v_h)} |\{k \in \mathbb{N}_0 : 0 \leq k < d(v) \text{ and } |\{j : i_j = k\}| \geq \gamma\}|.$$

Intuitively, $d'(v)$ counts the maximum number of "dense virtual distance levels" between v and t, where a vertex $v_j \in V'$ is allowed to occupy any one virtual distance level numbered at least $d(v_j)$, and where a dense virtual distance level is one that contains at least γ vertices in V'.

d' has the following properties:

(1) $\forall v \in V : 0 \leq d'(v) \leq h/\gamma$;

(2) $\forall v, w \in V : d(v) > d(w) \Longrightarrow d'(v) \geq d'(w)$;

(3) $\forall v, w \in V : (d(v) > d(w) \text{ and } |\{u \in V' : d(u) = d(w)\}| \geq \gamma) \Longrightarrow d'(v) > d'(w)$;

(4) a relabeling of a vertex $v \in V$ increases $d'(v)$ by at most 1 and does not increase $d'(w)$ for any $w \in V \backslash \{v\}$.

Define the potential function

$$\Phi = \sum_{v \in V - \{s,t\}} e(v) \cdot d'(v).$$

At the start of the Δ-phase, $\Phi \leq 2n\Delta \cdot h/\gamma$ (by property (1) and Fact 3.4), and $\Phi \geq 0$ always. Φ does not increase due to push operations (by property (2)), and a relabeling increases Φ by at most 3Δ (by property (4) and Fact 3.4). It follows that the total increase in Φ during the Δ-phase is at most $3\Delta \cdot D$. Consequently, the total decrease in Φ during the Δ-phase is at most $2n\Delta \cdot h/\gamma + 3\Delta \cdot D$. Finally note that each push satisfying the condition of the lemma and of value c causes Φ to decrease by at least c (by property (3)). □

Proof (Lemma 8.2). Consider a fixed Δ-phase. Let V' denote the set of vertices hit by the phase, let D denote the number of relabelings executed in the phase, and take $\gamma = \beta$. Now apply Lemma 8.3, noting that every nonterminal push satisfies the condition of the lemma. Hence, the ratio of the total value of all nonterminal pushes in the phase to $\Delta/2$ is $O(|V'| \cdot n/\beta + D)$.

The lemma now follows by summing over all phases, using Fact 7.5 to bound the sum over all phases of the number of vertices hit by the phase and noting that the total number of *relabel* operations is $O(n^2)$. □

The proof of the next lemma follows the same outline as that of Lemma 7.7.

LEMMA 8.4. $\sharp selects = O(nm/\beta + n^2\beta + n^2 \log n + \sharp \text{ptr})$.

Proof. First we give an improved bound on the number of *select* steps that execute either a *cut* or a *link* or a *relabel*, by giving an improved bound on the number of *cut* operations. Notice that the throughput of any saturating push associated with a *cut* operation is $\geq \Delta/2$, because whenever an edge (v, w) is inserted into F (the procedure *macropush*), then $rescap(v, w) \geq \Delta/2$ (by Fact 5.2). We claim that the number of medium saturating pushes with throughput $\geq \Delta/2$ is $O(nm/\beta + n^2\beta + n^2 \log n)$. To see this, focus on the nonterminal class (c) pushes with throughput $\geq \Delta/2$, since the total number of class (a), class (b), and terminal class (c) pushes is $O(nm/\beta + m \log n + n^2\beta)$, by Lemmas 7.1 and 8.1. By applying Lemma 8.2 it can be seen that the number of nonterminal class (c) pushes with throughput $\geq \Delta/2$ is $O(n^2 \log n)$. The claim follows. Consequently, the number of *cut* operations is $O(nm/\beta + n^2\beta + n^2 \log n + \sharp \textit{huge sat pushes} + \sharp \text{ptr}) = O(nm/\beta + n^2\beta + n^2 \log n + \sharp \text{ptr})$, and the number of *select* steps that execute either a *cut* or a *link* or a *relabel* is also $O(nm/\beta + n^2\beta + n^2 \log n + \sharp \text{ptr})$.

The remaining *select* steps are partitioned into two classes: those that execute one or more saturating pushes, and those that do not execute any saturating push. (The argument here is a refined version of that used in the proof of Lemma 7.7.)

The number of *select* steps that execute a saturating push and do not execute any *cut*, *link*, or *relabel* is $O(nm/\beta + n^2\beta + n^2 \log n + \sharp \textit{huge sat pushes})$. To see this, first focus on such a *select* step that executes a nonterminal class (c) push. Notice that all of the saturating pushes executed by the *select* step are nonterminal and that the total value of these pushes is $\geq \Delta/2$. Now, by applying Lemma 8.2, we see that over the whole execution, this case gives $O(n^2 \log n)$ *select* steps. Over the whole execution, the number of *select* steps that execute either a class (a), a class (b), or a terminal class (c) push is $O(nm/\beta + m \log n + n^2\beta)$, and the number of *select* steps under consideration that execute only small saturating pushes is $O(nm/\beta)$, because any such step executes at least $\beta/2$ saturating pushes, since the flow excess of the processed vertex decreases from $\geq \Delta$ to $< \Delta/2$ and since each small push has value $< \Delta/\beta$.

Now consider the *select* steps that do not execute any *cut*, *link*, *relabel*, or saturating push, and call each such step a *neat select* step. As noted before, the vertex v processed by a neat *select* step is a nonroot in F throughout that step. Consider a fixed vertex v and focus on the part of the execution between two consecutive nonneat *select* steps that process v, or after the last such *select* step. Suppose that v is processed by one or more neat *select* steps in this part of the execution. When the first of these neat *select* steps is executed, then $\Delta \leq e(v) < 3\Delta$. Hence, it follows by Fact 5.4 that $e(v)$ decreases to zero after at most two neat *select* steps that process v, and over the whole execution, this gives a number of neat *select* steps that is at most

twice the number of nonneat *select* steps. Before yet another neat *select* step that processes v, $e(v)$ has to increase from zero to $\geq \Delta$. Furthermore, by Fact 5.3, any increase of $e(v)$ must be caused by saturating pushes into v. Consider these saturating pushes and their associated edges (u, v), where $u \in V$. If each of these edges (u, v) has $ucap(u, v) < \Delta/\beta$ when v is processed by the neat *select* step, then clearly there are $\geq \beta$ saturating pushes into v, and over the whole execution this case gives $O(nm/\beta)$ neat *select* steps. Otherwise, either the status of one of these edges changes (from small to medium or huge) between the earliest of these saturating pushes and the neat *select* step that processes v, or there is at least one nonsmall saturating push into v. Over the whole execution, the former case gives $O(m)$ neat *select* steps, because the total number of status changes of edges is $\leq 2m$. If there is an increase of $e(v)$ from zero to $\geq \Delta$ due to a nonsmall saturating push and zero or more small saturating pushes, followed by a neat *select* step that processes v, then we have two mutually exclusive cases. Either the increase of $e(v)$ is due to one or more nonterminal class (c) pushes whose values sum to $\geq \Delta/2$ together with zero or more small saturating pushes, or the increase of $e(v)$ is due to one or more of the following together with zero or more small saturating pushes: one or more nonterminal class (c) pushes whose values sum to $< \Delta/2$ together with $\geq \beta/2$ small saturating pushes, a class (a) push, a class (b) push, a terminal class (c) push, or a huge saturating push. Over the whole execution, the first case gives $O(n^2 \log n)$ neat *select* steps (by Lemma 8.2), and the second case gives $O(nm/\beta + m \log n + n^2\beta + \sharp$*huge sat pushes*$)$ neat *select* steps. Therefore the total number of neat *select* steps is at most twice the total number of nonneat *select* steps plus $O(nm/\beta + n^2\beta + n^2 \log n + \sharp$*huge sat pushes*$)$.

Finally, the number of *select* steps, $\sharp$*selects*, is $O(nm/\beta + n^2\beta + n^2 \log n + \sharp\text{ptr})$, using the bound on $\sharp$*huge sat pushes* given by Lemma 6.1. □

THEOREM 8.5. *For every* $\alpha \geq 1$, *a maximum flow can be computed in time* $O(\alpha \cdot (nm + n^2(\log n)^2))$ *with probability at least* $1 - 2^{-\alpha(\sqrt{nm}+n \log n)}$.

Proof. By taking $\beta = 2 + \sqrt{m/n}$ and using the previous lemma and the bound on $\sharp$ptr given in Lemma 6.4, we see that $\sharp$*selects* $= O(\alpha \cdot (n^{3/2}m^{1/2} + n^2 \log n))$ with probability at least $1 - 2^{-\alpha(\sqrt{nm}+n \log n)}$. The bound on the running time follows from Lemma 5.5. □

Notice that for $m = \Omega(n(\log n)^2)$ the running time is $O(nm)$, with high probability.

Remark. If, as in [CH89], a new random permutation μ_v of $\Gamma(v)$ is computed at each relabeling of v, then the failure probability of Theorem 8.5 can be reduced even further to $2^{-\alpha(n^{3/2}m^{1/2}+n^2 \log n)}$.

9. Conclusion. We have shown that by using randomization on nonsparse graphs (i.e., $m = \Omega(n(\log n)^2)$) we can compute a maximum flow in $O(nm)$ time with high probability. The crucial new idea in our analysis is the notion of PTR events.

The preliminary version of this work [CH89] stimulated new research that has resulted in further advances on computing maximum flows. The original analysis is slightly loose, giving an $O(nm + n^{3/2}m^{1/2}(\log n)^{3/2} + n^2(\log n)^2)$ running time and an $O(n^{3/2}m^{1/2}(\log n)^{1/2} + n^2 \log n)$ bound on $\sharp$*selects*, both with high probability. Tarjan [T89] improved the analysis to obtain a running time of $O(nm + n^2(\log n)^2)$, with high probability. The improved analysis presented in §8, which differs from the analysis of [T89], was discovered subsequently and reported briefly in [CHM90].

Alon [A90] presented a simple way of derandomizing the strongly polynomial algorithm without affecting the running time for sufficiently dense graphs (i.e., $m = \Omega(n^{5/3} \log n)$): at initialization, the adjacency lists of the network vertices are permuted according to "pseudo-random permutations" (i.e., a set $\{\xi_1, \ldots, \xi_n\}$ of permutations of V with $\Lambda(\xi, \ldots, \xi_n) \ll n^2$) that are generated deterministically in $O(n^2)$ time. The running time of the resulting deterministic algorithm is $O(nm + n^{8/3} \log n)$. Recently, faster deterministic algorithms were described in [KRT93] and [PW93].

Another paper based on the present one is [CHM90], which shows that a maximum flow can be computed in $O(n^3/\log n)$ time on a uniform-cost RAM, and that the number of operations executed on flow variables can be improved from $O(nm)$ for the strongly polynomial algorithm here to $O(n^{8/3}(\log n)^{4/3})$.

One avenue for further research suggested by this work is to investigate whether randomization is useful for solving related problems on networks such as the minimum-cost flow problem and the maximum-weight matching problem.

REFERENCES

[AO89] R. K. AHUJA AND J. B. ORLIN, *A fast and simple algorithm for the maximum flow problem*, Oper. Res., 37 (1989), pp. 748–759.

[AOT89] R. K. AHUJA, J. B. ORLIN, AND R. E. TARJAN, *Improved time bounds for the maximum flow problem*, SIAM J. Comput., 18 (1989), pp. 939–954.

[A90] N. ALON, *Generating pseudo-random permutations and maximum flow algorithms*, Inform. Process. Lett., 35 (1990), pp. 201–204.

[CH89] J. CHERIYAN AND T. HAGERUP, *A randomized maximum flow algorithm*, in *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, 1989, pp. 118–123.

[CHM90] J. CHERIYAN, T. HAGERUP, AND K. MEHLHORN, *Can a maximum flow be computed in o(nm) time?*, in *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, 1990, Lecture Notes in Comput. Sci., 443 Springer-Verlag, New York, pp. 235–248.

[CM89] J. CHERIYAN AND S. N. MAHESHWARI, *Analysis of preflow push algorithms for maximum network flow*, SIAM J. Comput., 18 (1989), pp. 1057–1086.

[D70] E. A. DINIC, *Algorithm for solution of a problem of maximum flow in a network with power estimation*, Soviet Math. Dokl., 11 (1970), pp. 1277–1280.

[EK72] J. EDMONDS AND R. M. KARP, *Theoretical improvements in algorithmic efficiency for network flow problems*, J. Assoc. Comput. Mach., 19 (1972), pp. 248–264.

[FF57] L. R. FORD AND D. R. FULKERSON, *A simple algorithm for finding maximal network flows and an application to the Hitchcock problem*, Canad. J. Math., 9 (1957), pp. 210–218.

[FF62] ———, *Flows in Networks*, Princeton Univ. Press, Princeton, NJ, 1962.

[FT87] M. L. FREDMAN AND R. E. TARJAN, *Fibonacci heaps and their uses in improved network optimization algorithms*, J. Assoc. Comput. Mach., 34 (1987), pp. 596–615.

[F86] S. FUJISHIGE, *A capacity-rounding algorithm for the minimum-cost circulation problem: A dual framework of the Tardos algorithm*, Math. Programming, 35 (1986), pp. 298–308.

[Ga85] H. N. GABOW, *Scaling algorithms for network problems*, J. Comput. Systems Sci., 31 (1985), pp. 148–168.

[GaT88] Z. GALIL AND E. TARDOS, *An $O(n^2(m + n\log n)\log n)$ min-cost flow algorithm*, J. Assoc. Comput. Mach., 35 (1988), pp. 374–386.

[G85] A. V. GOLDBERG, *A new max-flow algorithm*, Tech. Rep. MIT/LCS/TM-291, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1985.

[GT88] A. V. GOLDBERG AND R. E. TARJAN, *A new approach to the maximum-flow problem*, J. Assoc. Comput. Mach., 35 (1988), pp. 921–940.

[GT89] ———, *Finding minimum-cost circulations by canceling negative cycles*, J. Assoc. Comput. Mach., 36 (1989), pp. 873–886.

[GT90] ———, *Finding minimum-cost circulations by successive approximation*, Math. Oper. Res., 15 (1990), pp. 430–466.

[GTT90] A. V. GOLDBERG, E. TARDOS, AND R. E. TARJAN, *Network flow algorithms*, in *Paths, Flows, and VLSI-Layout*, Algorithms and Combinatorics 9, B. Korte, L. Lovász, H. J. Prömel and A. Schrijver, eds., Springer-Verlag, Berlin, 1990, pp. 101–164.

[GLS88] M. GRÖTSCHEL, L. LOVÁSZ, AND A. SCHRIJVER, *Geometric Algorithms and Combinatorial Optimization*, Springer-Verlag, Berlin, 1988.

[K74] A. V. KARZANOV, *Determining the maximal flow in a network by the method of preflows*, Soviet Math. Dokl., 15 (1974), pp. 434–437.

[KRT93] V. KING, S. RAO, AND R. TARJAN, *A faster deterministic maximum flow algorithm*, preprint, January 1993; preliminary version in *Proceedings of the 3rd ACM-SIAM Symposium on Discrete Algorithms*, (1992), pp. 157–164; J. Algorithms, 17 (1994), pp. 447–474.

[O88] J. B. ORLIN, *A faster strongly polynomial minimum cost flow algorithm*, in Proceedings of the 20th ACM Symposium on Theory of Computing, 1988, pp. 377–387. Also in Sloan Working Paper No. 3060-89-MS, Massachusetts Institute of Technology, Cambridge, MA, 1989; Oper. Res., 41 (1993), pp. 338–350.

[PW93] S. PHILLIPS AND J. WESTBROOK, *Online load balancing and network flow*, in Proceedings of the 25th ACM Symposium on Theory of Computing, 1993, pp. 402–411.

[S77] R. SEDGEWICK, *Permutation generation methods*, Comput. Surv., 9 (1977), pp. 137–164.

[ST83] D. D. SLEATOR AND R. E. TARJAN, *A data structure for dynamic trees*, J. Comput. System Sci., 26 (1983), pp. 362–391.

[ST85] ———, *Self-adjusting binary search trees*, J. Assoc. Comput. Mach., 32 (1985), pp. 652–686.

[SV82] Y. SHILOACH AND U. VISHKIN, *An $O(n^2 \log n)$ Parallel MAX-FLOW Algorithm*, J. Algorithms, 3 (1982), pp. 128–146.

[T85] E. TARDOS, *A strongly polynomial minimum cost circulation algorithm*, Combinatorica, 5 (1985), pp. 247–255.

[T83] R. E. TARJAN, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[T89] ———, Personal communication, September 1989.

[T90] L. TUNÇEL, *On the complexity of preflow-push algorithms for maximum-flow problems*, Tech. Rep. 901, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, NY, April 1990; Algorithmica, 11 (1994), pp. 353–359.

SIAM J. COMPUT.
Vol. 24, No. 2, pp. 227–234, April 1995

002

SPARSE APPROXIMATE SOLUTIONS TO LINEAR SYSTEMS*

B. K. NATARAJAN†

Abstract. The following problem is considered: given a matrix A in $\mathbf{R}^{m\times n}$, (m rows and n columns), a vector b in $\mathbf{R}^m$, and $\epsilon > 0$, compute a vector x satisfying $\|Ax - b\|_2 \leq \epsilon$ if such exists, such that x has the fewest number of non-zero entries over all such vectors. It is shown that the problem is NP-hard, but that the well-known greedy heuristic is good in that it computes a solution with at most $\lceil 18\ \mathrm{Opt}(\epsilon/2)\|\mathbf{A}^+\|_2^2 \ln(\|b\|_2/\epsilon)\rceil$ non-zero entries, where $\mathrm{Opt}(\epsilon/2)$ is the optimum number of nonzero entries at error $\epsilon/2$, $\mathbf{A}$ is the matrix obtained by normalizing each column of A with respect to the L_2 norm, and $\mathbf{A}^+$ is its pseudo-inverse.

Key words. sparse solutions, linear systems

AMS subject classification. 68Q25

1. Introduction.

The problem of computing a sparse approximate solution to a linear system is a fundamental problem in matrix computation. For matrices over the reals, the problem (variant thereof) has been studied under the name "subset selection" in statistical modeling by Golub and Van Loan (1983). For binary matrices, the problem has been studied as "minimum weight solution" in error corrective coding by Gallager (1968). The related "sparse null-space" problem is of interest in nonlinear optimization; see Coleman and Pothen (1986). Also, the "minimum set cover" problem can be viewed as computing a sparse solution to $Ax \geq b$, where all the entries are binary and the inequality is entry-wise; see Garey and Johnson (1979), Johnson (1974).

Our specific motivation for studying the problem is as follows. A widely used technique for the interpolation of irregularly spaced samples in higher dimensions is that of radial basis interpolation; see Hardy (1988). In brief, assign each of the given sample points a coefficient. The value of the interpolant at a new point is the weighted sum of these coefficients, where the weights are the distances of the respective samples from the new point. The coefficients themselves are the solution to the linear system of equations obtained by evaluating the interpolant at the each of the given points and equating them to the respective sample values; Michelli (1986) has shown that the linear system is always nonsingular.

In solving the above linear system, two issues are of interest. (1) The cost of evaluating the interpolant at a general point grows with the number of nonzero coefficients in the solution. Hence, it is desirable to have as few nonzeros as possible, while tolerating some error in the interpolation. (2) By the principle of Occam's Razor, the interpolant with the fewest nonzero coefficients is most likely to approximate the underlying function that generated the samples. Indeed, in the presence of noise in the data, it is desirable to pick a sparse but approximate interpolant. Recently, learning theoretic justifications of Occam's Razor in this setting have been established; see Natarajan (1993).

In light of the above, we are interested in obtaining a sparse approximate solution to the linear system specifying the coefficients of the interpolant. Similar arguments can also be made with respect to other interpolation methods, such as polynomial or trigonometric interpolation.

Previously, it was known that the problem is NP-hard if the solution is required to be over the integers; see Garey and Johnson (1979). For matrices over the reals, the heuristic of Golub, Klema, and Stewart (1976) is well known, as discussed in Golub and Van Loan (1983). In this paper, we show that the problem remains NP-hard over the reals. We then show that the obvious and well-known greedy heuristic (Golub and Van Loan (1983)) is provably good in that

*Received by the editors November 2, 1992; accepted for publication October 29, 1993.
†Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, California 94304.

the number of nonzeros reported is at most 18 $\text{Opt}(\epsilon/2)\|\mathbf{A}^+\|_2^2 \ln(\|b\|_2/\epsilon)$, where $\text{Opt}(\epsilon/2)$ is the optimum number of nonzero entries at error $\epsilon/2$, $\mathbf{A}$ is the matrix obtained by normalizing each column of A with respect to the L_2 norm, and $\mathbf{A}^+$ is its pseudo-inverse. The algorithm is essentially a marriage between the greedy set cover algorithm of Johnson (1974) and the QR algorithm for least squares solutions of linear systems of Golub and Van Loan (1983). It is important to note that minimizing the number of zeros is competitive with the stability of the algorithm, i.e., the norm of the computed solution. The heuristic of Golub, Klema, and Stewart (1976) offers stability, but not provably good sparseness, while our algorithm does the converse. An open problem is to obtain an algorithm that allows an explicit tradeoff.

The algorithm consumes $O(mn)$ time per entry reported. Since the entries of the solution are reported in order of their "significance," the proposed algorithm is a good candidate for the progressive solution of dense linear systems. Computational experiments on using the greedy algorithm for radial basis interpolation are discussed in Carlson and Natarajan (1994).

2. Hardness. Let SAS refer to the sparse approximate solution problem stated below.

Problem. Given a matrix A in $\mathbf{R}^{m\times n}$, (m rows and n columns), a vector b in $\mathbf{R}^m$, and $\epsilon > 0$, compute a vector x satisfying $\|Ax - b\|_2 \leq \epsilon$ if such exists, such that x has the fewest number of nonzero entries over all such vectors.

We first establish that SAS is NP-hard on the machine model of the infinite precision RAM; see Preparata and Shamos (1985), for instance.

THEOREM 1. SAS *is* NP-*hard.*

Proof. The proof is by reduction from the problem of "exact cover by 3 sets," as in the proof of hardness for the problem of "minimum weight solutions to linear systems." See Garey and Johnson (1979), pp. 221 and 246).

Exact Cover by 3-*sets.*

Instance. A set S, and a collection C of 3-element subsets of S.

Question. Does C contain an exact cover for S, i.e., a sub-collection $\hat{C}$ of C such that every element of S occurs exactly once in $\hat{C}$?

Let us use X3C to refer to the problem of exact cover by 3-sets, and show how to transform an instance of X3C to an instance of SAS. Given is an instance of X3C: $S = \{s_1, s_2, \ldots, s_m\}$, and $C = c_1, c_2, \ldots, c_n$. Without loss of generality we can assume that m is a multiple of 3, since otherwise there is trivially no exact cover. Let b be the vector $(1, 1, 1, \ldots, 1)$ of m ones. The matrix A will have n column vectors, one for each set in C. Specifically, for each set $c \in C$, the corresponding column vector will have entries $(z_1, z_2, \ldots, z_m)$ where $z_i = 1$ if $s_i \in c$ and $z_i = 0$ otherwise. Pick $\epsilon = \frac{1}{2}$.

We show that the constructed instance of SAS has a solution with $m/3$ or fewer entries if and only if the given instance of X3C has a solution. If the X3C instance has a solution, then consider the vector $x = (x_1, x_2, \ldots, x_n)$, where $x_i = 1$ if the ith set in C is included in the solution of the X3C instance, and $x_i = 0$ otherwise. Then, $Ax = b$ exactly and hence the SAS instance has a solution with exactly $m/3$ nonzero entries. Conversely, assume the SAS instance has a solution x with at most $m/3$ nonzero entries.

Since $\|Ax - b\|_2 \leq \frac{1}{2}$, each entry of Ax must be between $\frac{1}{2}$ and $\frac{3}{2}$. Since each column of A has only three nonzero entries, x must have at least $m/3$ entries. Thus x has exactly $m/3$ entries. Now consider the subcollection $\hat{C}$, consisting of those sets c_i such that the ith entry in x is nonzero. It is clear that $\hat{C}$ is an exact cover for S. □

3. The algorithm. The algorithm we consider is a merging of the greedy set cover algorithm of Johnson (1974) and the QR algorithm for the least-squares problem of Golub and Van Loan (1983). In essence, it is a QR algorithm where the column pivot is chosen greedily with respect to the right-hand side b of the linear system. Such a modification is well known; see

Golub and Van Loan (1983, p. 168, Exercise 6.4.8). In the interest of simplicity, we will not present the algorithm as a modification of the QR algorithm, but will present it as a two-phase algorithm with a subset selection phase, followed by an explicit solution phase.

In words, Algorithm Greedy below takes as input A, b, and ϵ. The algorithm first normalizes each column a_i of A. Then, at each iteration of the selection phase, the algorithm greedily picks that column of A that is closest in angle to the vector b. Then, b and the column vectors of A are projected onto the subspace orthogonal to the chosen column. This procedure is repeated until $\|b\|_2 \leq \epsilon$. In the solution phase, the algorithm solves the linear system problem $Bx = b^{(0)} - b^{(r)}$ where B is the matrix consisting of those columns of A that were chosen in the selection phase, $b^{(0)}$ is b at the start of the selection of phase, and $b^{(r)}$ is b at the end of the selection of phase.

Algorithm Greedy
input:
matrix A, column vector b, $\epsilon > 0$.
Subset Selection Phase:
Normalize each column of A to obtain $\mathbf{A}$.
$r \leftarrow 0$; $\tau \leftarrow \phi$;
$A^{(0)} \leftarrow \mathbf{A}$; $b^{(0)} \leftarrow b$;
while $\|b^{(r)}\|_2 > \epsilon$ **do**
  if $A^{(r)T} b^{(r)} = 0$ **then**
    no solution exists;
  else
    choose $k \in \{1, 2, \ldots, n\} - \tau$ such that column $a_k^{(r)}$ of $A^{(r)}$
    is closest to $b^{(r)}$, i.e., $|a_k^{(r)T} b^{(r)}|$ is maximum;
    project $b^{(r)}$ onto the subspace orthogonal to $a_k^{(r)}$, i.e.,
    $b^{(r+1)} \leftarrow b^{(r)} - \left(a_k^{(r)T} b^{(r)}\right) a_k^{(r)}$;
    $\tau \leftarrow \tau \cup \{k\}$;
    for $j \in \{1, 2, \ldots, n\} - \tau$ **do**
      project $a_j^{(r)}$ onto the subspace orthogonal to $a_k^{(r)}$, i.e.,
      $a_j^{(r+1)} \leftarrow a_j^{(r)} - \left(a_k^{(r)T} a_j^{(r)}\right) a_k^{(r)}$;
      normalize $a_j^{(r+1)}$, i.e.,
      $a_j^{(r+1)} \leftarrow a_j^{(r+1)} / \|a_j^{(r+1)}\|_2$;
    end
  end
  $r \leftarrow r + 1$;
end
Solution Phase:
Solve $Bx = b^{(0)} - b^{(r)}$ where B is the matrix
of columns of A with indices in τ.
end

THEOREM 2. *The number of vectors selected by Algorithm Greedy, and hence the number of nonzero entries in the computed solution, is at most*

$$\left\lceil 18 \, \mathrm{Opt}(\epsilon/2) \|\mathbf{A}^+\|_2^2 \ln\left(\frac{\|b\|_2}{\epsilon}\right)\right\rceil, \tag{1}$$

where $\mathrm{Opt}(\epsilon/2)$ *denotes the fewest number of nonzero entries over all solutions that satisfy* $\|Ax - b\|_2 \leq \epsilon/2$.

Our proof of Theorem 2 is a simplified and generalized variant of that of the greedy set cover algorithm given by Johnson (1974). For each value of r, at the start of the corresponding iteration of the algorithm, let $u^{(r)}$ be a vector with the minimum number of nonzero entries such that $\|A^{(r)}u^{(r)} - b^{(r)}\|_2 \leq \epsilon/2$. Also, we define $N^{(r)}$ as the number of nonzero entries in $u^{(r)}$ and $q^{(r)} = A^{(r)}u^{(r)}$. Let t be the number of iterations the algorithm runs in the selection phase. It is clear that the number of nonzero entries produced by the solution phase of the algorithm is at most t. Define

$$\rho = 4 \max_{0 \leq r < t} \frac{N^{(r)}\|u^{(r)}\|_2^2}{\|b^{(r)}\|_2^2}. \tag{2}$$

Our proof proceeds in a sequence of lemmas. In the first lemma, we bound t in terms of ρ. Lemmas 2 and 3 estimate the right-hand side of the above definition of ρ. Combining the three lemmas, we will obtain Theorem 2.

For $z \in \mathbf{R}$, let $\lceil \mathbf{z} \rceil$ denote the smallest integer greater than or equal to z.

LEMMA 1. *The number of iterations of the algorithm t is at most*

$$\left\lceil 2\rho \ln\left(\frac{\|b\|_2}{\epsilon}\right)\right\rceil. \tag{3}$$

Proof. Since $u^{(r)}$ only has $N^{(r)}$ nonzero entries, we will show that at least one of the columns of $A^{(r)}$ must be correspondingly close to $b^{(r)}$. Specifically, we establish a lower bound on $\|A^{(r)T}b^{(r)}\|_\infty$. Now,

$$b^{(r)} = \sum_{i=1}^{n} u_i^{(r)} a_i^{(r)} + \epsilon\vec{/}2, \tag{4}$$

where $\epsilon\vec{/}2$ stands for the generic error vector of L_2 norm at most $\epsilon/2$. Hence

$$b^{(r)T}b^{(r)} = \sum_{i=1}^{n} u_i^{(r)} b^{(r)T} a_i^{(r)} + b^{(r)T}\epsilon\vec{/}2. \tag{5}$$

$$\leq \left(\max_{i=1}^{n} |\, a_i^{(r)T} b^{(r)} \,|\right) \sum_{i=1}^{n} |u_i^{(r)}| + \|b^{(r)}\|_2 \epsilon/2. \tag{6}$$

Now,

$$b^{(r)T}b^{(r)} = \|b^{(r)}\|_2^2, \tag{7}$$

$$\max_{i=1}^{n} |\, a_i^{(r)T} b^{(r)} \,| = \|A^{(r)T}b^{(r)}\|_\infty, \tag{8}$$

and

$$\|b^{(r)}\|_2 \geq \epsilon. \tag{9}$$

Also, since $u^{(r)}$ has $N^{(r)}$ nonzero entries,

$$\sum_{i=1}^{n} |u_i^{(r)}| = \|u^{(r)}\|_1 \leq \sqrt{N^{(r)}}\|u^{(r)}\|_2. \tag{10}$$

Hence, we can write (6) as

$$\|b^{(r)}\|_2^2 \leq \|A^{(r)T}b^{(r)}\|_\infty \sqrt{N^{(r)}}\|u^{(r)}\|_2 + 1/2\|b^{(r)}\|_2^2. \tag{11}$$

Rearranging, we get

$$\|A^{(r)T}b^{(r)}\|_\infty \geq \frac{\|b^{(r)}\|_2^2}{2\sqrt{N^{(r)}}\|u^{(r)}\|_2}. \tag{12}$$

Using the above lower bound on $\|A^{(r)T}b^{(r)}\|_\infty$ we now estimate the rate at which $b^{(r)}$ diminishes at each iteration of the algorithm. Now,

$$b^{(r+1)} = b^{(r)} - \left(a_k^{(r)T}b^{(r)}\right)a_k^{(r)}, \tag{13}$$

where $a_k^{(r)}$ is the column chosen by the algorithm at the rth iteration. Since $|a_k^{(r)T}b^{(r)}|$ is a maximum, $|a_k^{(r)T}b^{(r)}| = \|A^{(r)T}b^{(r)}\|_\infty$ and we can rewrite the above as

$$b^{(r+1)} = b^{(r)} - \text{sign}\left(a_k^{(r)T}b^{(r)}\right)\|A^{(r)T}b^{(r)}\|_\infty a_k^{(r)}, \tag{14}$$

where sign( ) is the function that returns the sign of its argument. Since $b^{(r+1)}$ is orthogonal to $a_k^{(r)}$,

$$\|b^{(r+1)}\|_2^2 = \|b^{(r)}\|_2^2 - \|A^{(r)T}b^{(r)}\|_\infty^2. \tag{15}$$

Rearranging, we get

$$\|b^{(r+1)}\|_2^2 = \left(1 - \frac{\|A^{(r)T}b^{(r)}\|_\infty^2}{\|b^{(r)}\|_2^2}\right)\|b^{(r)}\|_2^2. \tag{16}$$

Substituting (12) in (16), we get

$$\|b^{(r+1)}\|_2^2 \leq \left(1 - \frac{\|b^{(r)}\|_2^2}{4N^{(r)}\|u^{(r)}\|_2^2}\right)\|b^{(r)}\|_2^2. \tag{17}$$

Using (2) in (17), we get

$$\|b^{(r+1)}\|_2^2 \leq (1 - 1/\rho)\|b^{(r)}\|_2^2. \tag{18}$$

Hence,

$$\|b^{(t)}\|_2^2 \leq (1 - 1/\rho)^t\|b^{(0)}\|_2^2. \tag{19}$$

The algorithm halts when $\|b^{(t)}\|_2^2 \leq \epsilon^2$. For this to happen, it suffices that t satisfy

$$(1 - 1/\rho)^t\|b^{(0)}\|_2^2 \leq \epsilon^2. \tag{20}$$

Rearranging and taking logarithms, we get

$$t\ln(1 - 1/\rho) \leq \ln\left(\frac{\epsilon^2}{\|b^{(0)}\|_2^2}\right). \tag{21}$$

Incorporating in (21) the following inequality, obtained from a Taylor expansion of the natural logarithm function,

$$\ln(1 - 1/\rho) \leq \frac{-1}{\rho}, \tag{22}$$

we get that it suffices if

$$t \geq \rho \ln\left(\frac{\|b^{(0)}\|_2^2}{\epsilon^2}\right) = 2\rho \ln\left(\frac{\|b\|_2}{\epsilon}\right). \tag{23}$$

This completes the proof of the lemma. □

We will now establish an upper bound on ρ, by first bounding $\|u^{(r)}\|_2$ and then incorporating a bound on $N^{(r)}$.

LEMMA 2. *For* $0 \leq r < t$, $\|u^{(r)}\|_2 \leq 3/2\|\mathbf{A}^+\|_2\|b^{(r)}\|_2$.

Proof. Let $\sigma = \{i \mid u_i^{(r)} \neq 0\}$ be the set of indices of the nonzero entries in $u^{(r)}$, and let $\tau = \{k_1, k_2, \ldots, k_r\}$ be the indices of the first r columns picked by the algorithm. For matrix A and set of indices μ, we use $\mu(A)$ to denote the set of column vectors $\{a_i \mid i \in \mu\}$.

We first show that $\sigma(A^{(0)}) \bigcup \tau(A^{(0)})$ is a linearly independent set. Assume the contrary, i.e., some linear combination of vectors from $\sigma(A^{(0)}) \bigcup \tau(A^{(0)})$ sum to zero. Not all of the vectors in this combination can be from $\tau(A^{(0)})$, since $\tau(A^{(0)})$ is a linearly independent set by the definition of the algorithm. So it must be that at least one of the vectors in this combination is from $\sigma(A^{(0)})$. It follows that this vector from $\sigma(A^{(0)})$ can be expressed as a linear combination of vectors from $\sigma(A^{(0)}) \bigcup \tau(A^{(0)})$. Since $\sigma(A^{(r)})$ is the projection of $\sigma(A^{(0)})$ on the subspace orthogonal to $\tau(A^{(0)})$, the above supposition implies that some vector in $\sigma(A^{(r)})$ can be expressed as a linear combination of vectors in $\sigma(A^{(r)})$. But this contradicts the assumption that $u^{(r)}$ has a minimum number of nonzero entries. Hence $\sigma(A^{(0)}) \bigcup \tau(A^{(0)})$ must be linearly independent.

We can now establish a bound on $\|u^{(r)}\|_2$. By definition,

$$q^{(r)} = \sum_{i \in \sigma} u_i^{(r)} a_i^{(r)}. \tag{24}$$

For $i \in \sigma$, we can write

$$a_i^{(r)} = \frac{a_i^{(r-1)} - v_i^{(r)}}{\sqrt{1 - \|v_i^{(r)}\|_2^2}}, \tag{25}$$

where $v_i^{(r)}$ is a vector in the span of $\tau(A^{(r-1)})$. Using this recurrence, we can express $a_i^{(r-1)}$ in terms of $a_i^{(r-2)}$ and so on down to $a_i^{(0)}$. Since $\tau(A^{(r)})$ is orthogonal to $\tau(A^{(r-1)})$, it follows that $v_i^{(r)}$ is orthogonal to $v_i^{(r-1)}$ and hence $\|v_i^{(r)} + v_i^{(r-1)}\|_2^2 = \|v_i^{(r)}\|_2^2 + \|v_i^{(r-1)}\|_2^2$. Using this, we can combine the recurrences (25) to obtain a single expression of the form

$$a_i^{(r)} = \frac{a_i^{(0)} - v_i}{\sqrt{1 - \|v_i\|_2^2}}, \tag{26}$$

where v_i is a vector in the span of $\tau(A^{(0)})$.

Substituting (26) in (24), we get

$$q^{(r)} = \sum_{i \in \sigma} u_i^{(r)} \frac{a_i^{(0)} - v_i}{\sqrt{1 - \|v_i\|_2^2}}. \tag{27}$$

Noting that v_i is a vector in the span of $\tau(A^{(0)})$ and hence can be expressed as a linear combination of the columns of $\tau(A^{(0)})$, we can rewrite (27) as

$$q^{(r)} = \sum_{i \in \sigma} \frac{u_i^{(r)}}{\sqrt{1 - \|v_i\|_2^2}} a_i^{(0)} + \sum_{i \in \tau} \delta_i a_i^{(0)}, \tag{28}$$

for some δ_i in $\mathbf{R}$. Let Z be the matrix with columns $\sigma(A^{(0)}) \bigcup \tau(A^{(0)})$, and Z^+ its pseudo-inverse. Also, let $w = Z^+ q^{(r)}$. As established above, the columns of Z are linearly independent, and hence the entries of w are uniquely determined as in (28) above. Specifically, for $i \in \sigma$,

$$w_i = \frac{u_i^{(r)}}{\sqrt{1 - \|v_i\|_2^2}}, \tag{29}$$

and for $i \in \tau$, $w_i = \delta_i$. Since $\sqrt{1 - \|v_i\|_2^2} < 1$, we have that for $i \in \sigma$, $|u_i^{(r)}| \leq |w_i|$. For $i \in \tau$, $w_i = \delta_i$, $u_i^{(r)} = 0$, and hence $|u_i^{(r)}| \leq |w_i|$. We therefore have

$$\|u^{(r)}\|_2 \leq \|w\|_2 \leq \|Z^+\|_2 \|q^{(r)}\|_2. \tag{30}$$

Noting that $q^{(r)} = b^{(r)} + \epsilon \vec{/} 2$, and that since $r < t$, $\|b^{(r)}\|_2 > \epsilon$, we can simplify the above as

$$\|u^{(r)}\|_2 \leq \|Z^+\|_2 \|b^{(r)} + \epsilon \vec{/} 2\|_2 \leq 3/2 \|Z^+\|_2 \|b^{(r)}\|_2. \tag{31}$$

Since the columns of Z are a linearly independent subset of the columns of $A^{(0)} = \mathbf{A}$, the smallest nonzero singular value of Z is greater than or equal to the smallest nonzero singular value of $\mathbf{A}$; see Golub and Van Loan (1983, p. 286). Recall that $\mathbf{A}^+$ is the pseudo-inverse of $\mathbf{A}$. Since $\|Z^+\|_2$ and $\|\mathbf{A}^+\|_2$ are the reciprocals of the smallest nonzero singular values of Z and $\mathbf{A}$, respectively, $\|Z^+\|_2 \leq \|\mathbf{A}^+\|_2$, and the proof of the lemma is complete. □

LEMMA 3. *The number of entries in the sparse solution is nonincreasing as the algorithm iterates, i.e.,* $N^{(r+1)} \leq N^{(r)} \leq N^{(0)}$.

Proof. As before let $\sigma = \{i \mid u_i^{(r)} \neq 0\}$ be the set of indices of the nonzero entries in $u^{(r)}$, By definition, $|\sigma| = N^{(r)}$. Suppose that the algorithm selects a vector from σ at the rth iteration. Then surely $u^{(r+1)}$ has one fewer nonzero entry than $u^{(r)}$, namely the entry corresponding to the selected vector. If the vector selected by the algorithm is not from σ, then σ remains the set of nonzero entries of $u^{(r+1)}$. In either case $N^{(r+1)} \leq N^{(r)}$ and the lemma follows. □

Noting that $N^{(0)} = \text{Opt}(\epsilon/2)$ we can use Lemma 3 to rewrite (2) as

$$\rho \leq 4 \max_{0 \leq r < t} \frac{N^{(0)} \|u^{(r)}\|_2^2}{\|b^{(r)}\|_2^2}. \tag{32}$$

Using Lemma 2 to substitute for $\|u^{(r)}\|_2$ in (32) gives us

$$\rho \leq 9 \max_{0 \leq r < t} N^{(0)} \|A^{(+)}\|_2^2. \tag{33}$$

Substituting (33) in the statement of Lemma 1, we get that the number of zeros in the computed solution is at most

$$\left\lceil 18 \, \text{Opt}(\epsilon/2) \|\mathbf{A}^+\|_2^2 \ln\left(\frac{\|b\|_2}{\epsilon}\right) \right\rceil. \tag{34}$$

This concludes the proof of Theorem 1.

Acknowledgments. Thanks to C. Bischof, R. Carlson, T. Coleman, J. Gilbert, D. S. Johnson, and J. Ruppert for the discussions, to A. Frieze for helping clear an error in an earlier version of the paper, and to the anonymous referees for their comments.

REFERENCES

R. E. CARLSON AND B. K. NATARAJAN (1994), *Sparse approximate multiqaudric interpolation*, Comp. and Math. Appl., Vol. 27 (6), pp. 99–108.

T. COLEMAN AND A. POTHEN (1986), *The null space problem* I. *Complexity*, SIAM J. Alg. Disc. Meth., Vol. 7, pp. 527–537.

R. G. GALLAGER (1968), *Information Theory and Reliable Communication*, John Wiley, New York, NY.

M. R. GAREY AND D. S. JOHNSON (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, New York.

G. H. GOLUB, V. KLEMA, AND, G. W. STEWART (1976), *Rank Degeneracy and Least Squares Problems*, Tech. Report, TR-456, Dept. of Computer Science, University of Maryland, College Park, MD.

G. H. GOLUB AND C. F. VAN LOAN (1983), *Matrix Computations*, Johns Hopkins Press, Baltimore, MD.

R. L. HARDY (1990), *Theory and applications of the multi-quadric biharmonic method*, Comput. Math. Appl., Vol. 19, pp. 163–208.

D. S. JOHNSON (1974), *Approximation algorithms for combinatorial problems*, J. Comput. System Sci., Vol. 9, pp. 256–278.

C. A. MICHELLI (1986), *Interpolation of scattered data: distance matrices and conditionally positive definite systems*, Constr. Approx., Vol. 2, pp. 11–22.

B. K. NATARAJAN (1993), *Occam's razor for functions*, Proceedings of the ACM Symposium on Computational Learning Theory, Santa Cruz, CA.

F. P. PREPARATA AND M. I. SHAMOS (1985), *Computational Geometry, An Introduction*, Springer-Verlag, New York.

SIAM J. COMPUT.
Vol. 24, No. 2, pp. 235–241, April 1995

003

A HEURISTIC SCHEDULING OF INDEPENDENT TASKS WITH BOTTLENECK RESOURCE CONSTRAINTS*

QINGZHOU WANG† AND ERNST L. LEISS†

Abstract. In this paper, the problem of scheduling independent tasks with bottleneck resource constraints is investigated. There is a set of independent tasks $\mathcal{T} = \{T_1, \ldots, T_n\}$ and a set of resources $\mathcal{R} = \{R_1, \ldots, R_m\}$. The available amount of resource R_j, for $j = 1, \ldots, m$, is (normalized to) 1. Each task T_i's execution requires at least $\lambda_{(i,j)} \leq 1$ units of resource $R_j \in \mathcal{R}$, and with these minimal resources, the execution time of T_i is τ_i. If there is a resource $R_{\beta(i)}$ so that T_i can use ξ units ($\xi \geq \lambda_{(i,\beta(j))}$) and reduce the execution time to $\frac{\lambda_{(i,\beta(i))}}{\xi}\tau_i$, we say that T_i can achieve *linear speed-up* with respect to $R_{\beta(i)}$. It is assumed that for each T_i, there is one, and only one, resource $R_{\beta(i)}$ that is T_i's *bottleneck* resource: T_i can use ξ units of $R_{(i,\beta(i))}$ and achieve linear speed-up, where ξ is between $\lambda_{(i,\beta(i))}$ and $\Lambda_{(i,\beta(i))} \leq 1$. The problem is to find a feasible schedule for all the tasks in $\mathcal{T}$ that has a shortest overall makespan.

This problem is a combination of two previously studied problems—scheduling tasks with resource constraints [*SIAM J. Comput.*, 4 (1975), pp. 187–200.] scheduling parallel tasks [*SIAM J. Comput.*, 21 (1992), pp. 281–294]. A variant of this problem was studied in [*J. Combin. Theory*, 21 (1976), pp. 257–298]. Because the problem is NP-hard, we propose the ECT (*earliest completion time*) algorithm as a heuristic solution and show that the performance ratio of the ECT makespan M_{ECT} to the optimal makespan M_{OPT} is bounded by $2l + m + 1$, where l is the number of the bottleneck resources in $\mathcal{R}$. When $l = 0$, this is exactly the performance bound shown by Garey and Graham in [*SIAM J. Comput.*, 4 (1975), pp. 187–200].

Key words. bottleneck resource, heuristic, and independent task scheduling

AMS subject classifications. 68Q20, 68Q25

1. Introduction.

In the problem of scheduling independent tasks with resource constraints, we have a set $\mathcal{R}$ of m resources, $R_1, \ldots, R_m$, and a set $\mathcal{T}$ of n independent tasks, $T_1, \ldots, T_n$. For each resource R_j, the total amount available is always (normalized to) 1. For each task T_i, there is a minimum resource requirement for its execution—T_i requires $\lambda_{(i,j)} \leq 1$ units of resource R_j, $j = 1, \ldots, m$. With these minimal resources, task T_i can be completed in τ_i time units. For each T_i, there is one, and only one, *bottleneck* resource $R_{\beta(i)}$. In other words, the function

$$\beta: \quad \{1, \ldots, n\} \to \{1, \ldots, m\}, \tag{1.1}$$

maps a task index to a resource index. A resource is said to be a bottleneck resource for T_i if T_i can use up to $\Lambda_{(i,\beta(i))} \leq 1$ units of resource $R_{\beta(i)}$ and achieve a linear speed-up; if T_i uses ξ units of resource $R_{\beta(i)}$, where $\lambda_{(i,\beta(i))} \leq \xi \leq \Lambda_{(i,\beta(i))}$, and $\lambda_{(i,j)}$ units of all other resources $R_j (j \neq \beta(i))$, the required time to complete T_i will be shortened to $(\lambda_{(i,\beta(i))}/\xi)\tau_i$. If a task T_i uses $\Lambda_{i,\beta(i))}$ units of its bottleneck resource during the execution, we call it *fully parallelized.*

If a resource R_j is a bottleneck resource of some task, we simply call it a *bottleneck resource*. Not all resources need to be bottleneck resources, so the number of bottleneck resources is denoted by l, with $0 \leq l \leq m$.

An execution schedule of n tasks in $\mathcal{T}$ is called *feasible*, if at any time during the schedule, the usage of each individual resource R_j, $j = 1, \ldots, m$ does not exceed the available amount, 1. Among all the feasible schedules, it is desirable to use the optimal schedule, i.e., that which completes all the tasks in the shortest possible time. The time to complete all the tasks is called *makespan*; the optimal (shortest possible) makespan is denoted by M_{OPT}. Finding the optimal schedule is NP-hard [3]. Therefore, we investigate heuristic algorithms that take polynomial time to implement but produce only suboptimal schedules. If S is a heuristic scheduling

*Received by the editors January 25, 1993; accepted for publication (in revised form) November 9, 1993.
†Department of Computer Science, University of Houston, Houston, Texas 77204-3475.

algorithm, we use M_S to denote its makespan. The closeness of how a heuristic algorithm S approximates the optimal scheduling is expressed by its performance bound: $\sup\{M_S/M_{\mathrm{OPT}}\}$.

In this paper, we investigate the performance of using the *earliest completion time* (ECT) algorithm [4] to schedule independent tasks with bottleneck constraints. We will show that in general the performance bound of the ECT algorithm is

$$\sup\left\{\frac{M_{\mathrm{ECT}}}{M_{\mathrm{OPT}}}\right\} = 3m + 1. \tag{1.2}$$

This can be refined to $2l + m + 1$ if there are only l bottleneck resources. In discussing the independent task scheduling with no bottleneck resources (i.e., $l = 0$), Garey and Graham [1] use a scheduling algorithm that is a special case of ECT and whose performance bound is $m + 1$. Furthermore, in a variant of this problem, where each task is assumed to always have unit execution time, Garey et al. [2] develop scheduling algorithms based on the first fit (FF) and the first fit decreasing (FFD) bin packing techniques. The FF and FFD scheduling algorithms have asymptotic performance bounds of $m + \frac{7}{10}$ and $m + \frac{1}{3}$, respectively.

The discussion in this paper will be as follows. The ECT algorithm and its graphical representation in the multiresource context will be presented first. Following the graphical representation, we will introduce the notion of *covering intervals*, which is based on those of *wide* and *narrow* tasks that were first defined in [4]. Then, we analyze the resource usage and derive our performance bound.

2. The ECT algorithm. The algorithm in the top half of Fig. 1 is a LIST algorithm in its simplest form, and without loss of generality, we can assume that the task index reflects the execution order according to the LIST scheduling. Because each task has its minimum resource requirements, the time t_N is the time when there are sufficient resources to execute at least one waiting task. Immediately starting the next task T_i, however, does not mean that we can give T_i its earliest completion time. For each task T_i ready to be executed,

```
input ( task set 𝒯 )
while 𝒯 ≠ ∅ do
    t_N ← time when there are sufficient resources for at least one T_i
    invoke Earliest_Completion(t_N, T_i)
    𝒯 ← 𝒯 − {T_i}
end_while
```

```
    procedure Earliest_Completion(t_N, T_i)
    for ξ ∈ [λ_(i,β(i))), Λ_(i,β(i))]{
        define t(ξ) = the earliest time ≥ t_N there are ξ units of R_β(i)
        define t̃(ξ) = t(ξ) + (λ_(i,β(i)) / ξ) τ_i
    }
(†) ξ_i ← min{t̃^{-1}(min{t̃(ξ) : ∀ξ})};
    schedule T_i with ξ_i units of R_β(i) at time t(ξ_i);
```

FIG. 1. *The ECT Algorithm for tasks with bottleneck resources.*

we further process it through the *Earliest_Completion* procedure as shown in the second half of Fig. 1. If more bottleneck resource $R_{\beta(i)}$ is becoming available soon, we are willing to delay the starting time of T_i to achieve its earliest completion.

LEMMA 2.1. *The $T()$ function defined in the Earliest_Completion procedure only assumes no more than n values.*

Proof. Let us assume that $T_{i_1}, \ldots, T_{i_k}$ are the running tasks at time t_N, and T_{i_l} completes before $T_{i_{l+1}}$. If T_{i_l} releases ξ_l units of $R_{\beta(i)}$ at its completion time, we can write the total

amount of $R_{\beta(i)}$ released by T_i and the tasks before it with the notation $\xi_l = \sum_0^l \zeta_j$. For all $\xi \in (\xi_l, \xi_{l+1}]$, $t(\xi)$ will equal the completion time of T_{i_l}.

When all T_{i_l}'s have completed, $\sum_0^k \zeta_j = 1$ and $R_{\beta(i)}$ is completely free. Since all T_{i_l}'s only form a subset of $\mathcal{T}$, i.e., $k+1 \leq n$, we have actually shown that $t()$ assumes no more than n values in $[\lambda_{(i,\beta(i))}, 1]$, so it can assume no more than n values in $[\lambda_{(i,\beta(i))}, \Lambda_{(i,\beta(i))}]$. □

LEMMA 2.2. *The value* $\min\{\tilde{t}(\xi) : \forall \xi \in [\lambda_{(i,\beta(i))}, \Lambda_{(i,\beta(i))}]\}$ *in the statement* (†) *in Fig.* 1 *can be calculated in no more than n steps.*

Proof. In the previous lemma, we have shown that $t()$ assumes no more than n values in the interval $(\xi_l, \xi_{l+1}]$. By the definition of $\tilde{t}$, within each such interval, $\tilde{t}()$ is a strictly descending function that attains the minimal value at ξ_{l+1}. Therefore the calculation of min value in (†) in Fig. 1 can be done in no more than n steps. □

LEMMA 2.3. *In the statement* (†) *in Fig.* 1, $\tilde{t}^{-1}(\min\{\tilde{t}(\xi) : \forall \xi \in [\lambda_{(i,\beta(i))}, \Lambda_{(i,\beta(i))}]\})$ *may not be a unique value, yet it will never assume more than n values.*

Proof. As shown in the proof of Lemma 2.2, $\tilde{t}()$ is strictly descending in the interval $(\xi_l, \xi_{l+1}]$. Since there are no more than $k+1$ such intervals, the inverse function in (†) in Fig. 1 has no more than $k+1 \leq n$ values. □

There are four time events associated with the scheduling of a task T_i:

(i) *preparation time*, $a_S(T_i) = t(\lambda_{i,\beta(i))})$;

(ii) *execution time*, $b_S(T_i) = t(\xi_i)$;

(iii) *completion time*, $c_S(T_i) = \tilde{t}(\xi_i)$; and

(iv) *full parallelization time*, $f_S(T_i) = \min\{t(\Lambda_{(i,\beta(i))}), c_S(T_i)\}$.

The following relationships should always hold:

$$(2.1) \qquad a_S(T_i) \leq b_S(T_i) < f_S(T_i) \leq c_S(T_i) \quad \text{and} \quad c_S(T_i) - b_S(T_i) = \frac{\lambda_{(i,\beta(i))}}{\xi_i}\tau_i,$$

where ξ_i is the actual number of units of $R_{\beta(i)}$ used by T_i.

3. The graphical representation. Each resource can be represented by a strip of height 1; before it is used by the tasks, the strip is white from time 0 to ∞. Without loss of generality, let us consider the graphic representation of a single resource R_j. R_j is a strip of height 1 with the following coloring patterns:

(i) R_j is the bottleneck resource of T_i, so in $[a_S(T_i), b_S(T_j))$, T_i's preparation introduces grey color (Fig. 2(a));

(ii) otherwise, R_j is colored *black*, completely or partially, representing the amount of resource being used (Fig. 2(b)).

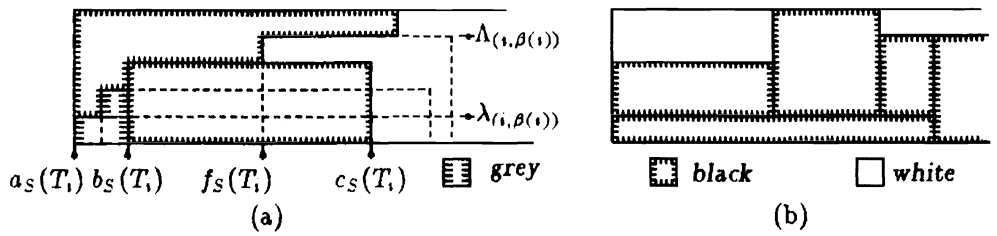


FIG. 2. *Two coloring patterns of a resource strip.*

LEMMA 3.1. *When it is colored in the first pattern, the strip* R_j *within the interval* $a_S(T_i), b_S(T_i))$ *is completely colored black or grey.*

Proof. This is really a direct result of the ECT scheduling. If there were any white space (free resource), it would have been consumed during the preparation of task T_i, and thus become grey. □

4. Wide and narrow tasks. Similar to [4], we put each task into one of two categories, $\mathcal{T} = \mathcal{T}_W \cup \mathcal{T}_N$:

(i) $T_i \in \mathcal{T}_W$, the set of *wide* tasks, if $\Lambda_{(i,\beta(i))} \geq \frac{1}{2}$;

(ii) otherwise, $T_i \in \mathcal{T}_N$, the set of *narrow* tasks.

We further define the *covering interval* $[X(T_i), Z(T_i))$ associated with each task T_i:

$$\begin{aligned} &X(T_i) = a_S(T_i), \qquad Z(T_i) = c_S(T_i), \quad \text{if } T_i \in \mathcal{T}_W; \\ &X(T_i) = a_S(T_i), \qquad Z(T_i) = f_S(T_i), \quad \text{if } T_i \in \mathcal{T}_N. \end{aligned} \tag{4.1}$$

If we eliminate from $\mathcal{T}$ the task $T_{i'}$ that satisfies

$$\exists i < i', \text{ such that } Z(T_{i'}) \leq Z(T_i), \tag{4.2}$$

we will have a subsequence of tasks

$$\mathcal{T}' = \{T_{i_1}, \ldots, T_{i_{n'}}\}. \tag{4.3}$$

The tasks in $\mathcal{T}'$ satisfy the following condition:

$$Z(T_{i_1}) < \cdots < Z(T_{i_k}) < \cdots < Z(T_{i_{n'}}). \tag{4.4}$$

This leads to a set of covering intervals that do not overlap:

$$[Y(T_{i_k}), Z(T_{i_k})) : \begin{cases} \text{if } k = 1, & Y(T_{i_k}) = X(T_{i_k}), \\ \text{otherwise}, & Y(T_{i_k}) = \max\{X(T_{i_k}), Z(T_{i_{k-1}})\}. \end{cases} \tag{4.5}$$

Associated with each such covering interval, there is a unique resource $R_{\beta(i_k)}$ that is the bottleneck resource of T_{i_k}. The next lemma estimates the resource usage of $R_{\beta(i_k)}$ in the interval $[Y(T_{i_k}), Z(T_{i_k}))$.

LEMMA 4.1. *Within the interval* $[y(T_{i_k}), Z(T_{i_k}))$, *the usage of the resource* $R_{\beta(i_k)}$ *is at least* $\frac{1}{2}(Z(T_{i_i}) - Y(T_{i_k}))$.

Proof. Let us consider the usage of $R_{\beta(i_k)}$ by all tasks $T_i \in \mathcal{T}$, with $i \leq i_k$. We distinguish the following two cases separately:

(i) the task T_{i_k} is a wide task;

(ii) the task T_{i_k} is a narrow task.

In the first case, we can directly use the analysis of wide tasks in [4, Lem. 6] after normalizing the unit of available resource to 1. It follows that the strip $R_{\beta(i_k)}$ in the interval is at least half colored black.

In the second case, T_{i_k} is a narrow task, and the right end of the interval is $Z(T_{i_k}) = f_S(T_{i_k}) \leq t(\Lambda_{i_k,\beta(i_k)}))$, which is no later than the time when task T_{i_k} could be fully parallelized If a narrow task cannot be fully parallelized, no more than half of the resource can be unused. Lemma 3.1 and Fig. 2(a) can help us to illustrate the argument. We can also conclude in the second case that the usage of $R_{\beta(i_k)}$ is no less than $\frac{1}{2}(z(T_{i_k}) - Y(T_{i_k}))$. □

In analyzing the usage of resources, there is a fundamental difference between the LIST scheduling of sequential tasks and the ECT scheduling of parallel tasks.

(i) In sequential task scheduling, for every task, the amount of resource it uses is always fixed, independent of the scheduling.

(ii) In ECT scheduling, the amount of bottleneck resource used by a task is always fixed as shown in the linear speed-up. The usage of nonbottleneck resource R_j by T_i can vary from $\lambda_{(i,j)}\tau_i$ to $\lambda_{(i,j)}\ \lambda_{(k,\beta(i))}/\Lambda_{(i,\beta(i))}\ \tau_i$, where the later is the minimal amount. The actual usage is scheduling dependent.

Because of this dependency, we must show how much $R_{\beta(i_k)}$ has been used by T_i, other than T_{i_k}, as a nonbottleneck resource.

LEMMA 4.2. *If $T_i, i < i_k$, uses $R_{\beta(i_k)}$ as a nonbottleneck resource, its usage within $[Y(T_{i_k}), Z(T_{i_k}))$ is minimal, not exceeding $\lambda_{(i,\beta(i_k))}\, \lambda_{(i,\beta(i))}/\Lambda_{(i,\beta(i))}\, \tau_i$.*

Proof. We are considering any $T_i \in \mathcal{T}$, where $\beta(i) \neq \beta(i_k)$ and $i < i_k$. However, we can immediately eliminate any wide task T_i, because $Z(T_i)$ equals its completion time, and this occurs before $Y(T_{i_k})$.

For a narrow task, because of the inequality of $Z(T_i) = \min\{c_S(T_i), t(\Lambda_{(i,\beta(i))})\} \leq Y(T_{i_k})$, whether $T_k \in \mathcal{T}'$ or $T_i \in \mathcal{T} - \mathcal{T}'$ we know that at time $Y(T_{i_k})$, T_i either has completed or could have been fully parallelized on its own bottleneck resource. Therefore, its usage of $R_{\beta(i_k)}$ is no greater than $\lambda_{(i,\beta(i_k))}\, \lambda_{i,\beta(i))}/\Lambda_{i,\beta(i))}\, \tau_i$, the minimal required amount. □

The following proposition reveals the relationship between the total length of $\cup[Y(T_{i_k}), Z(T_{i_k}))$ and M_{OPT}.

PROPOSITION 4.1. *If there are l bottleneck resources, $1 \leq l \leq m$, the combined length of all nonoverlapping intervals, $\sum(Z(T_{i_k}) - Y(T_{i_k}))$ will not exceed $2l \cdot M_{\mathrm{OPT}}$.*

Proof. After constructing the intervals (4.5), each interval is uniquely associated with a resource $R_{\beta(i_k)}$, which is at least half utilized by those tasks T_i starting before T_{i_k} and T_{i_k} itself (at least half colored black if it is considered as a strip). We have further shown that if a task T_i uses $R_{\beta(i_k)}$ as a nonbottleneck resource, its usage will not exceed the minimal amount. Therefore, the total length of intervals (4.5) associated with a single bottleneck resource will be no greater than $2M_{\mathrm{OPT}}$. Since there are only l bottleneck resources, the proposition follows. □

5. The overall analysis. After analyzing the intervals (4.5), we cannot claim that the complete makespan $[0, M_{\mathrm{ECT}})$ is covered by these intervals. Therefore, we must investigate the intervals of

$$[0, M_{\mathrm{ECT}}) - \bigcup_{k=1}^{n'} [Y(T_{i_k}), Z(T_{i_k})) = \bigcup_{i=1}^{\tau} [f_i, g_i). \tag{5.1}$$

At this point, we need a graph theory based result explained in [1]. There, the optimal makespan is normalized to 1 when compared with the suboptimal makespan to derive the performance bound. Without loss of generality, we also assume $M_{\mathrm{OPT}} = 1$. By using the same notation $r_j(t)$ to represent the utilization of resource R_j at time t, the following lemma, first appearing in [1], also holds for our ECT algorithm.

LEMMA 5.1. *If $t_0, t_1 \in \cup_{i=1}^{r}[f_i, g_i)$, and $t_1 - t_0 \geq 1$,*

$$\max_{j=1}^{m}\{r_j(t_0) + r_j(t_1)\} > M_{\mathrm{OPT}} = 1$$

will hold.

Proof. The ECT selection criteria is the same as that of the LIST algorithm, i.e, selecting any task that the available resources will satisfy. Since $t_1 - t_0 \leq M_{\mathrm{OPT}} = 1$, there is at least one task T_i that starts between $(t_0, t_1]$. If the statement is not true, the task T_i should have been scheduled by the ECT algorithm at t_0 or sooner. This is the same lemma as shown in [1], and the argument is also the same. □

Now we can apply the theorem based on graph theory from [1] and reach the conclusion that

$$\sum_{i=1}^{r}(g_i - f_i) \leq m + 1 = (m+1)M_{\mathrm{OPT}}. \tag{5.2}$$

Again, we encounter the fundamental difference between the LIST scheduling of sequential tasks and the ECT scheduling of parallel task, i.e., the usage of nonbottleneck resource is dependent on the ECT scheduling.

LEMMA 5.2. *If in* $[f_i, g_i)$, *a task* T_k *uses a nonbottleneck resource* R_j, *the total amount used will not exceed* $\lambda_{(k,j)}\ \lambda_{(k,\beta(k))}/\Lambda_{(k,\beta(k))}\ \tau_k$. *In other words, despite the difference mentioned earlier, the usage of nonbottleneck resources in* $\cup[f_i, g_i)$ *is always minimal.*

Proof. Based on the definition of intervals (4.5), only some narrow task could have part of their execution extended into intervals of $\cup[f_i, g_j)$. By the definition of $Z(T_{i_k})$, such tasks extending into $\cup[f_i, g_i)$ have passed the point of achieving full parallelization when entering $\cup[f_i, g_i)$. Using the same argument of Lemma 4.2, we can show the usage of nonbottleneck resources in $\cup[f_i, g_i)$ is also minimal. □

We have recognized the difference between the conventional LIST scheduling and the ECT scheduling. Despite this difference, in $\cup[f_i, g_i)$, the usage of nonbottleneck resources is always kept minimal as long as the ECT scheduling criterion is observed. This leads to the adaptation of the following proposition from the LIST scheduling analysis to our analysis.

PROPOSITION 5.1. *The total length of* $\cup[f_i, g_i)$ *will not exceed* $m + 1 = (m + 1)M_{\mathrm{OPT}}$, *i.e., the inequality of* (5.2) *will hold.*

Proof. The proof in [1] relies on two results, namely, Lemma 5.1 and the same resource usage in the proposed scheme as that in the optimal scheme. After getting the total usages in both the optimal and the proposed schemes, it can be shown that the total usage $\leq m$ in the optimal case implies that the total usage in the proposed case will not exceed m.

We have already proved our Lemma 5.1. As for the total usage, the usage of bottleneck resources does not change between the optimal schedule and the ECT schedule. The usage of nonbottleneck resources is minimal within $\cup[f_i, g_i)$; it could be even less than the usage in the optimal schedule within these intervals. We can conclude that within $\cup[f_i, g_i)$ the total usage will not exceed m, so the proposition follows. □

Combining Propositions 4.1 and 5.2, we can have the following theorem.

THEOREM 5.1. *The performance bound of the ECT algorithm for scheduling tasks with bottleneck resources satisfies the following inequality:*

$$\frac{M_{\mathrm{ECT}}}{M_{\mathrm{OPT}}} \leq 2l + m + 1, \tag{5.3}$$

where l *is the number of bottleneck resources.*

6. Conclusion. In this paper, we have discussed the earliest completion time scheduling of tasks requiring multiple resources and among which one is a bottleneck resource. There are two special cases of this problem that have been studied previously.

(i) There is no bottleneck resource, $l = 0$. Garey and Graham [1] have shown that $m + 1$ is a tight performance bound that is achieved by our algorithm.

(ii) There is only one resource, $m = 1$, and it is also the bottleneck resource, i.e., $l = 1$. In [4], we have shown that the ECT performance bound is at least 2.5. There is a gap of 1.5 between the performance bound 2.5 and the performance bound of $2l + m + 1 = 4$ shown in the present paper This gap is wider than the gap ($\approx \frac{1}{2}$) shown in [4], for the current analysis is relaxed to accommodate the requirement of multiple resources. As pointed out in [1], the constant term 1 in $2l + m + 1$ is needed only in some special instances, and this term accounts for the gap difference.

In the analysis of the FFD algorithm for scheduling tasks with unit execution time [2], it was shown that its asymptotic performance bound lies between $m + (m - 1)/m(m + 1)$ and $m + \frac{1}{3}$. In other words, the performance bound of this scheduling algorithms is dominated by the m term. In the ECT algorithm, the term $2l$ can assume a wide range of values between 0 and $2m$

depending on the number of bottleneck resources involved. It will be interesting to see how l, the number of bottleneck resources, can further influence the analysis and the tightness of the performance bound.

REFERENCES

[1] M. R. GAREY AND R. L. GRAHAM, *Bounds for multiprocessor scheduling with resource constraints*, SIAM J. Comput., 4 (1975), pp. 187–200.

[2] M. R. GAREY, R. L. GRAHAM, D. S. JOHNSON, AND A. C. YAO, *Resource constrained scheduling as generalized bin packing*, J. Combin. Theory, 21 (1976), pp. 257–298.

[3] R. L. GRAHAM, E. L. LAWLER, J. K. LENSTRA, AND A. H. G. RINNOOY KAN, *Optimization and Approximation in Deterministic Sequencing and Scheduling: A survey*, Ann. Discrete Math., 15 (1979), pp. 287–326.

[4] Q. WANG AND K. H. CHENG, *A heuristic of scheduling parallel tasks and its analysis*, SIAM J. Comput., 21 (1992), pp. 281–294.

SIAM J. COMPUT.
Vol. 24, No. 2, pp. 242–246, April 1995

004

ON COMPUTING ALGEBRAIC FUNCTIONS USING LOGARITHMS AND EXPONENTIALS*

DIMA GRIGORIEV,† MICHAEL SINGER,‡ AND ANDREW YAO§

Abstract. Let ρ be a set of algebraic expressions constructed with radicals and arithmetic operations, and which generate the splitting field F of some polynomial. Let $N_\beta(\rho)$ be the minimum total number of root-takings and exponentiations used in any straightline program for computing the functions in ρ by taking roots, exponentials, logarithms, and performing arithmetic operations. In this paper it is proved that $N_\beta(\rho) = \nu(G)$, where $\nu(G)$ is the minimum length of any cyclic Jordan–Hölder tower for the Galois group G of F. This generalizes a result of Ja'Ja' [*Proceedings of the* 22nd *IEEE Symposium on Foundations of Computer Science*, 1981, pp. 95–100], and shows that the inclusion of certain new primitives, such as taking exponentials and logarithms, does not improve the cost of computing such expressions as compared with programs that use only root-takings.

Key words. algebraic expressions, complexity, differential algebra, exponentials, Galois group, Jordan–Hölder tower, logarithms

AMS subject classifications. 68Q25, 68Q40

1. Introduction.

The question of how efficiently one can evaluate expressions such as $(\Sigma_{1\le i<j\le n}\sqrt{(x_i-x_j)^2+(y_i-y_j)^2})/\binom{n}{2}$, the *mean distance* among n points in the plane, was raised in Shamos and Yuval [9]. A systematic study of this question was given in Pippenger [7], [8]. Let ρ be a family of algebraic expressions constructed from indeterminates using radicals and arithmetic operations. Define the cost of a program to be the number of root-takings used, with arithmetic operations given for free. Let $\mathbf{F}$ be the extension field generated by the members of ρ over the field of rational functions with complex coefficients. It was shown [7], [8] that, when the members of ρ are rational functions of the roots of rational functions, the minimum cost is equal to the number of the torsion orders for the Galois group of $\mathbf{F}$ (an Abelian group in this case). An extension was given[1] in Ja'Ja' [2], who showed that the minimum cost is equal to the minimum length of any cyclic Jordan–Hölder tower for the Galois group of $\mathbf{F}$, provided that $\mathbf{F}$ is a finite Galois extension over the field of rational functions. It is known [2], [8] that the former result is a special case of the latter.

These results can be used to determine the minimum cost for computing ρ in many cases. For example, for the mean distance problem, the Galois group can be shown [7] to be isomorphic to $(\mathbf{Z_2})^{\binom{n}{2}}$, which clearly has $\binom{n}{2}$ torsion orders.

As taking a root $y^{1/d}$ can be simulated by taking the logarithm $\log y$ followed by an exponentiation $\exp((\log y)/d)$, a natural question is whether the availability of the logarithm and exponential operations can substantially reduce the cost of evaluating algebraic expressions. In particular, can one evaluate the expression $\Sigma_{1\le i\le n}\sqrt{x_i}$ using $o(n)$ exponentiations and logarithm-takings? (Clearly, the expression can be evaluated with n root-takings.) The possible use of logarithms and exponentials, as well as other primitives, was mentioned in [9], but was not studied in later papers [2], [7], [8].

*Received by the editors February 22, 1993; accepted for publication November 9, 1993.

†Departments of Computer Science and Mathematics, Pennsylvania State University, University Park, Pennsylvania 16802.

‡Department of Mathematics, North Carolina State University, Raleigh, North Carolina 27695-8205. The research of this author was supported in part by National Science Foundation grant DMS-90-24624.

§Department of Computer Science, Princeton University, Princeton, New Jersey 08544. The research of this author was supported in part by the National Science Foundation under grant CCR-9301430.

[1] Any finite Abelian group G can be uniquely decomposed into a direct sum of cyclic groups $\mathbf{Z_{d_1}} \oplus \mathbf{Z_{d_2}} \oplus \cdots \mathbf{Z_{d_t}}$, such that $d_t > 1$ and d_i is divisible by d_{i+1} for $1 \le i < t$. The integers d_i are called the *torsion orders* for G; t is the number of torsion orders for G.

In this paper, we show that under the same assumption as in [2] (i.e., $\mathbf{F}$ being a finite Galois extension), the availability of taking logarithms and exponentials does not reduce the cost. In particular, we prove that n or more operations are needed to evaluate $\Sigma_{1\le i\le n}\sqrt{x_i}$, with arithmetic operations given for free. In the next section, we give a precise statement of the main result (Theorem 1), after introducing the needed notation and background. The result is then proved in §3; some additional concepts and results from differential algebra (see [3]–[5]) are used in the proof.

We remark that the complexity question under other cost measures, in which the cost of taking a dth root may depend on d, was discussed in [2], [7], [8]. We will not pursue it here. We also would like to mention a related recent paper by Grigoriev and Karpinski [1], in which another complexity question involving root-takings was studied.

2. The main result. We use the standard terminology in algebra (as in Lang [6]). In what follows, let $\mathbf{Z}^+$ be the set of all positive integers.

An *α-program* A is a sequence of instructions of the form $z_1 \leftarrow I_1, z_2 \leftarrow I_2, \ldots, z_m \leftarrow I_m$, where I_i are of the form $(r_i(x_1, x_2, \ldots, x_n, z_1, \ldots, z_{i-1}))^{1/d_i}$ with r_i being any rational function in $x_1, \ldots, x_n, z_1, \ldots, z_{i-1}$ with complex coefficients and $d_i \in \mathbf{Z}^+$. We call m the *cost* of A. For $1 \le i \le m$, let $g_i(x_1, x_2, \ldots, x_n)$ be the functions defined inductively by $g_i(x_1, x_2, \ldots, x_n) = (r_i(x_1, x_2, \ldots, x_n, g_1(x_1, \ldots, x_n), \ldots, g_{i-1}(x_1, \ldots, x_n)))^{1/d_i}$. We always assume that the r_i have been chosen so that the denominators of these functions do not vanish identically. Informally, $g_i(x_1, x_2, \ldots, x_n)$ are the values assumed by the variables z_i for input $(x_1, x_2, \ldots, x_n)$. Let E_A denote the set of all functions of the form $r(x_1, x_2, \ldots, x_n, g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$ where r is any rational function with complex coefficients whose denominator does not vanish identically when the substitution is made.

We note that each element of E_A defines a function algebraic over the field $\mathbf{F_0} = \mathbf{C}(x_1, \cdots, x_n)$ of rational functions in n variables with coefficients in the complex numbers $\mathbf{C}$.

A *solvable algebraic expression* is any element of E_A for any α-program A. Let $\rho = (f_1, f_2, \ldots, f_s)$ be a finite set of solvable algebraic expressions. We say that *ρ is computed by* A, if each $f_i \in E_A$. Let $N_\alpha(\rho)$ be the minimum cost of any α-program computing ρ. Clearly, $N_\alpha(\rho)$ is finite. For any such ρ, we can form the field $\mathbf{F_0}(\rho)$, which is the algebraic extension of $\mathbf{F_0}$ formed by adjoining the functions corresponding to the elements $f_1, \ldots, f_s$ of ρ.

Following [2], ρ is said to be *normal*, if $\mathbf{F_0}(\rho)$ is a finite Galois extension of $\mathbf{F_0}$. In other words, ρ is normal if ρ generates the splitting field of some polynomial over $\mathbf{F_0}$.

For any solvable group G, a *cyclic Jordan–Hölder tower* is a normal tower of groups

$$G = G_0 \triangleright G_1 \triangleright \cdots \triangleright G_{m-1} \triangleright G_m = 1,$$

where G_{i-1}/G_i is cyclic for each $1 \le i \le m$. Let $\nu(G)$ be the length m of the shortest cyclic Jordan-Hölder tower for G.

The next result is from Ja'Ja' [2], which we state as a lemma.

LEMMA 1 (See [2]). *If ρ is normal, then $N_\alpha(\rho) = \nu(G)$, where G is the Galois group for $\mathbf{F_0}(\rho)$ over $\mathbf{F_0}$.*

A *β-program* B is a sequence of instructions of the form $z_1 \leftarrow I_1, z_2 \leftarrow I_2, \ldots, z_m \leftarrow I_m$, where I_i are of the form a_i^{1/d_i}, $\exp(a_i)$, or $\log(a_i)$, where $a_i = r_i(x_1, x_2, \ldots, x_n, z_1, \ldots, z_{i-1})$ with r_i being any rational function in $x_1, \ldots, x_n, z_1, \ldots, z_{i-1}$ with complex coefficients and $d_i \in \mathbf{Z}^+$. We again always assume that the r_i have been chosen so that the denominators of these functions do not vanish identically. Let $\tau(B)$ be the number of instructions that either take roots or exponentials. Let $g_i(x_1, x_2, \ldots, x_n)$ be the functions associated with variables z_i, defined exactly as in the case for α-programs. Let E_B denote the set of all functions of the form $r(x_1, x_2, \ldots, x_n, g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$ where r is any rational function

with complex coefficients whose denominators do not vanish identically when the substitution is made.

Let $\rho = (f_1, f_2, \cdots, f_s)$ be a finite set of solvable algebraic expression. We say that ρ *is computed by* B if each element f_i of ρ equals a function in E_B. Let $N_\beta(\rho)$ be the minimum $\tau(B)$ of any β-program B computing ρ.

Our main result is the following theorem.

THEOREM 1. *If ρ is normal, then $N_\beta(\rho) = \nu(G)$, where G is the Galois group for $\mathbf{F_0}(\rho)$ over $\mathbf{F_0}$.*

COROLLARY 1. *If ρ is normal, then $N_\beta(\rho) = N_\alpha(\rho)$.*

COROLLARY 2. *Let $\rho = \{f\}$, where $f = \Sigma_{1\le i\le n}\sqrt{x_i}$. Then $N_\beta(\rho) = n$.*

Remark. It is an interesting open question whether $N_\beta(\rho)$ is equal to $N_\alpha(\rho)$ when ρ is not required to be normal.

3. Proof of Theorem 1. Before proving the theorem, we introduce some terms in differential algebra (see [3]–[5]). A *differential field* is a field k together with a set $\Delta = \{\delta_i\}$ of mappings $\delta_i : k \to k$, called *derivations*, such that each δ_i satisfies the conditions $\delta_i(a + b) = \delta_i(a) + \delta_i(b)$, $\delta_i(ab) = \delta_i(a)b + a\delta_i(b)$, and $\delta_i(\delta_j(a)) = \delta_j(\delta_i(a))$ for all $\delta_i, \delta_j \in \Delta, a, b \in k$. For example, $\mathbf{F_0}$ can be considered a differential field when we use the derivations $\Delta = \{\delta_1, \ldots, \delta_n\}$ where $\delta_i(f) = \partial f/\partial x_i$. In this paper we are concerned only with differential fields that come from fields of differentiable functions a and that are extensions of this differential field. These extensions will be gotten by adjoining elements that can be interpreted as functions on some suitable region in complex n-space $\mathbf{C}^n$. We will use $\mathbf{K_0}$ to denote the differential field obtained from the field $\mathbf{F_0}$ equipped with these standard derivations Δ. Note that if $\mathbf{K}$ is a differential field containing $\mathbf{K_0}$ and if $a \in \mathbf{K}$, then the field obtained by adjoining $\exp(a)$ to $\mathbf{K}$ gives a differential field. The element $\exp(a)$ will satisfy the differential equations $\delta_i(\exp(a)) = \delta_i(a) \cdot \exp(a)$ for $i = 1, \ldots, n$. Similarly, the adjoining of $\log(a)$ gives a differential field and the element $\log(a)$ satisfies $\delta_i(\log(a)) = \delta_i(a)/a$ for $i = 1, \ldots, n$. We also note that if $\rho = (f_1, \ldots, f_s)$ is a set of solvable algebraic expressions (or, more generally, any set of algebraic functions), the derivations Δ can be extended uniquely to derivations on $\mathbf{F_0}(\rho)$ ([5, Lem. 1, p. 90]).

The classical Galois theory for field theory can be extended to a *differential Galois theory* for differential fields (See [4] and [5] for definitions and discussions of these concepts; [3] contains an excellent exposition of the theory in the case of only one derivation and the essential results extend, mutatis mutandi, to the case of several derivations). This Galois theory can be used to study the structure of the solutions of a system of partial linear differential equations, provided that the equations generate a differential ideal of finite linear dimension or, equivalently (see [5, Chap. IV.5]), the solution space is a finite-dimensional vector space (i.e., the system is holonomic). This is the case for the equations defining exponentials and logarithms (see [4] and [5]). To avoid possible confusion, we will reserve the term *Galois group* for the classical Galois group, and use the term *differential* Galois group when differential fields are being discussed. It should be noted, however, that if k_1 is an algebraic extension of k_0, a differential field of characteristic zero, then since all derivations on k_0 can be extended uniquely to derivations on k_1, we can identify the Galois group of k_1 over k_0 with the differential Galois group of k_1 over k_0 (with respect to these derivations). To see this note that any differential automorphism is by definition a usual automorphism. Conversely, for any automorphism σ of k_1 and k_0 and any derivation δ of k_1 that leaves k_0 invariant, we have that $\sigma^{-1} \circ \delta \circ \sigma$ is a derivation of k_1 agreeing with δ on k_0. Uniqueness implies that they must be equal on all of k_1 and so σ must be a differential automorphism. This remark allows us to apply results concerning differential Galois theory to the Galois theory of algebraic extensions of differential fields.

To prove Theorem 1, we first show that if $\mathbf{F_0}(\rho)$ is contained in a certain tower of differential fields, then there is a tower of algebraic extension fields of no greater length containing $\mathbf{F_0}(\rho)$. This result (Lemma 2) is at the heart of the proof for Theorem 1.

Let $\mathbf{K_0} \subset \mathbf{K_1} \subset \mathbf{K_2} \subset \cdots \subset \mathbf{K_m}$ be a tower of differential fields, where each $\mathbf{K_i}$ is obtained from $\mathbf{K_{i-1}}$ by adjoining an element u_i; u_i is either $\exp(a_i)$ or $\log(a_i)$ with $a_i \in \mathbf{K_{i-1}}$. Let I be the set of $1 \leq i \leq m$ such that u_i is $\exp(a_i)$. We recall from differential Galois theory that in this case each $\mathbf{K_i}$ is a Picard–Vessiot extension of $\mathbf{K_{i-1}}$. Furthermore, it is known (see [4, §4], or [5, Chap. VI.6]; [3, Lem. 3.9 and 3.10] contains similar results for the case of one derivation) that, if $i \in I$, the differential Galois group of $\mathbf{K_i}$ over $\mathbf{K_{i-1}}$ is an algebraic subgroup of $\mathbf{C}^*$, the multiplicative group of nonzero complex numbers, and if $i \notin I$, then the differential Galois group of $\mathbf{K_i}$ over $\mathbf{K_{i-1}}$ is an algebraic subgroup of $\mathbf{C}^+$, the additive group of complex numbers. Finally, we note that the proper algebraic subgroups of $\mathbf{C}^*$ are precisely the finite cyclic groups and the only proper algebraic subgroup of $\mathbf{C}^+$ is the trivial group. This can be seen by noting that a proper Zariski closed subset of either of these two groups must be finite and that in the first case, we will have a finite multiplicative subgroup of a field and in the second case we will have a finite subgroup of a torsion free group.

LEMMA 2. *If* $\mathbf{F_0}(\rho) \subset \mathbf{K_m}$ *then* $\nu(G) \leq |I|$.

Proof. Let $\mathbf{F_i} = \mathbf{F_0}(\rho) \cap \mathbf{K_i}$ for $1 \leq i \leq m$. Then $\mathbf{F_m} = \mathbf{F_0}(\rho)$. Note that $\mathbf{F_0} = \mathbf{F_0}(\rho) \cap \mathbf{K_0}$. Let H_i be the differential Galois group of $\mathbf{K_i}$ over $\mathbf{K_{i-1}}$. We claim that the following statement is true for $1 \leq i \leq m$.

FACT 1. $\mathbf{F_i}$ *is a Galois extension of* $\mathbf{F_{i-1}}$.

To prove this fact, let $\mathbf{E_i}$ be the subfield of elements of $\mathbf{K_i}$ algebraic over $\mathbf{K_{i-1}}$. $\mathbf{E_i}$ is a differential field and is left invariant by all elements of H_i. Therefore the differential Galois group of $\mathbf{K_i}$ over $\mathbf{E_i}$ is a normal subgroup of H_i and so $\mathbf{E_i}$ is a Galois extension of $\mathbf{K_{i-1}}$. Note that $\mathbf{F_i} = \mathbf{E_i} \cap \mathbf{F_0}(\rho)$. Let $p(x)$ be a polynomial with coefficients in $\mathbf{F_{i-1}}$. If $p(x) = 0$ has a root in $\mathbf{F_i}$, it must split in both $\mathbf{E_i}$ and $\mathbf{F_0}(\rho)$ (since $\mathbf{F_0}(\rho)$ is a fortiori normal over $\mathbf{F_{i-1}}$). Therefore $p(x) = 0$ splits in $\mathbf{F_i}$ and so $\mathbf{F_i}$ is a Galois extension of $\mathbf{F_{i-1}}$.

Now let J_i be the Galois group of $\mathbf{F_i}$ over $\mathbf{F_{i-1}}$. We claim that the following statement is true.

FACT 2. For $1 \leq i \leq m$, J_i *is the trivial group if* $i \notin I$, *and a cyclic group if* $i \in I$.

To prove this fact, consider the field $\mathbf{K_{i-1}} \cdot \mathbf{F_i}$. This is a subfield of $\mathbf{K_i}$. Since H_i is an abelian group, all of its subgroups are normal, so $\mathbf{K_{i-1}} \cdot \mathbf{F_i}$ is a normal extension of $\mathbf{K_{i-1}}$ whose differential Galois group L_i is the quotient of H_i by a closed subgroup of H_i. Furthermore, since $\mathbf{K_{i-1}} \cdot \mathbf{F_i}$ is a finite extension of $\mathbf{K_{i-1}}$, L_i is finite and thus coincides with the Galois group of this extension. If $i \notin I$, then H_i is either $\mathbf{C}^+$ or the trivial group. The only finite quotient of either of these groups by a closed subgroup is trivial. If $i \in I$, then H_i is either $\mathbf{C}^*$ or a finite cyclic group. The only possible finite quotients of these groups by closed subgroups are cyclic. To finish the proof of Fact 2, we note that $\mathbf{K_{i-1}} \cap \mathbf{F_i} = \mathbf{F_{i-1}}$ and so the Galois group J_i of $\mathbf{F_i}$ over $\mathbf{F_{i-1}}$ is isomorphic to L_i (see [6, Cor., p. 400] or [5, Chap. VII, Th. 1.12]; [3, Lem. 5.10] gives a related result but deals only with the case of one derivation).

We can now finish the proof of Lemma 2. Let G_i denote the group of automorphisms of $\mathbf{F_0}(\rho)$ leaving $\mathbf{F_i}$ fixed. By Facts 1 and 2, we conclude from the Galois theory that the series $G = G_0, G_1, G_2, \ldots, G_m = 1$ forms a cyclic Jordan–Hölder tower, with G_{i-1}/G_i being isomorphic to J_i. Deleting all $i \notin I$, we have a tower of length $|I|$. Hence $\nu(G) \leq |I|$. □

We now turn to the proof of Theorem 1. Observe that $N_\beta(\rho) \leq N_\alpha(\rho)$, which is not greater than $\nu(G)$ by Lemma 1. Thus, we only need to prove that $N_\beta(\rho) \geq \nu(G)$.

Let B be any β-program for computing ρ. Without loss of generality, we may assume that no root-taking operations are used in B, as we can replace any instruction $z \leftarrow r^{1/d}$ by two instructions $y \leftarrow (\log r)/d, z \leftarrow \exp(y)$ without changing the value of $\tau(B)$. Let the

instructions be $z_1 \leftarrow I_1, z_2 \leftarrow I_2, \ldots, z_m \leftarrow I_m$. Let $g_i(x_1, x_2, \ldots, x_n)$ be the functions associated with variables z_i.

For $1 \leq i \leq m$, let $\mathbf{K_i}$ be the differential field obtained by adjoining g_i to $\mathbf{K_{i-1}}$. By definition, the functions of E_B correspond to elements of $\mathbf{K_m}$ and $\mathbf{F_0}(\rho) \subset \mathbf{K_m}$. By Lemma 2, this implies $\nu(G) \leq \tau(B)$. This proves $\nu(G) \leq N_\beta(\rho)$ and completes the proof of Theorem 1. Corollary 1 follows immediately from the theorem and Lemma 1.

To prove Corollary 2, we note that ρ is normal and the Galois group G of $\mathbf{F_0}(\rho)$ over $\mathbf{F_0}$ is isomorphic to $\mathbf{Z_2}^n$. From the results in [2], [8] (see [8, Lem. 3.2, p. 399]), $\nu(G)$ is equal to the number of torsion orders of G, which is clearly n. Corollary 2 follows from the theorem immediately.

REFERENCES

[1] D. Grigoriev and M. Karpinski, *Computation of the Additive Complexity of Algebraic Circuits with Root Extracting*, Tech. Report, TR-92-079, International Computer Science Institute, Berkeley, CA, 1992.

[2] J. Ja' Ja', *Computation of algebraic functions with root extractions*, Proceedings of 22nd IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1981, pp. 95–100.

[3] I. Kaplansky, *An Introduction to Differential Algebra*, Hermann, Paris, 1957.

[4] E. R. Kolchin, *Picard–Vessiot theory of partial differential fields*, Proc. Amer. Math. Soc., 3 (1952), pp. 596–603.

[5] E. R. Kolchin, *Differential Algebra and Algebraic Groups*, Academic Press, New York, 1973.

[6] S. Lang, *Algebra*, 2nd ed., Addison-Wesley, Menlo Park, CA, 1984.

[7] N. Pippenger, *Computational complexity of algebraic functions*, J. Comput. System Sci., 22 (1981), pp. 454–470.

[8] N. Pippenger, *Corrections to "Computational complexity of algebraic functions"*, J. Comput. System Sci., 37 (1988), pp. 395–399.

[9] M. Shamos and G. Yuval, *Lower bounds from complex function theory*, Proceedings of 17th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1976, pp. 268–273.

SIAM J. COMPUT.
Vol. 24, No. 2, pp. 247–265, April 1995

005

ALGORITHMS FOR ENUMERATING ALL SPANNING TREES OF UNDIRECTED AND WEIGHTED GRAPHS*

SANJIV KAPOOR† AND H. RAMESH‡

Abstract. In this paper, we present algorithms for enumeration of spanning trees in undirected graphs, with and without weights.

The algorithms use a search tree technique to construct a computation tree. The computation tree can be used to output all spanning trees by outputting only relative changes between spanning trees rather than the entire spanning trees themselves. Both the construction of the computation tree and the listing of the trees is shown to require $O(N + V + E)$ operations for the case of undirected graphs without weights. The basic algorithm is based on swapping edges in a fundamental cycle. For the case of weighted graphs (undirected), we show that the nodes of the computation tree of spanning trees can be sorted in increasing order of weight, in $O(N \log V + VE)$ time. The spanning trees themselves can be listed in $O(NV)$ time. Here N, V, and E refer, respectively, to the number of spanning trees, vertices, and edges of the graph.

Key words. spanning tree, undirected graph, weighted graph, enumeration

AMS subject classifications. 68Q25, 68R10

1. Introduction.

Spanning tree enumeration in undirected graphs is an important issue in many problems encountered in network and circuit analysis. Applications are given in [Ma72]. Weighted spanning tree enumeration in order would find application in a subroutine of a generate-and-test procedure for connecting together a set of points with the minimum amount of wire, where the connection satisfies some additional constraint, e.g., a minimum distance to be maintained between two wires.

Spanning tree enumeration has a long history (see references). Previous techniques employed for solving the problem include depth-first search [GM78], [TR75], selective generation and testing [Ch68], and edge exchanging [Ga77]. Of these, Gabow and Myers' algorithm [GM78] seems to be the fastest with a time complexity of $O(NV)$ on a graph with V vertices, E edges, and N spanning trees. Their algorithm requires $O(NV)$ time for generating the trees themselves and not merely for outputting them. Their algorithm is optimal, up to a constant factor, if all spanning trees of the graph need to be explicitly output. For many practical applications, the spanning trees need not be explicitly output and only a computation tree which gives relative changes between spanning trees is required. We note that from this computation tree, the spanning trees can be listed explicitly in $O(NV)$ operations, if required.

In this paper, we enumerate spanning trees by listing differences between them. Each node of the computation tree that describes this procedure represents a spanning tree of the graph. The spanning trees represented by a node and its parent in the computation tree differ in exactly one pair of edges, i.e., the spanning tree at any node is obtained by exchanging an edge in the spanning tree at its parent for an edge not present in that spanning tree. This exchange is obtained from the fundamental cycles of the graph. An edge external to a spanning tree can be exchanged with any edge in its fundamental cycle to give a spanning tree which differs from the original spanning tree in exactly one pair of edges. By repeating this for all external edges, all spanning trees which differ from the original spanning tree in one pair of edges

*Received by the editors April 30, 1992; accepted for publication (in revised form) June 4, 1993. An extended abstract of this work appeared in the proceedings of the Workshop on Algorithms and Data Structures 91, Ottawa, Lecture Notes in Comput. Sci. 519, Springer-Verlag, New York.

†Department of Computer Science and Engineering, Indian Institute of Technology, Hauz Khas, New Delhi,India (skapoor@cse.iitd.ernet.in).

‡Department of Computer Science and Engineering, Indian Institute of Technology, Hauz Khas, New Delhi, India. Present address, Courant Institute of Mathematical Sciences, New York University, New York, N.Y. 10012 (ramesh@slinky.cs.nyu.edu).

can be obtained. The computation tree is generated by repeatedly applying this procedure. Repetition of the same spanning tree is avoided by following a search tree inclusion–exclusion strategy. The algorithm presented here outputs, for each node of the computation tree, the difference between the spanning trees associated with that node and its predecessor in a preorder scan of the computation tree. This is done by traversing the computation tree in a depth-first manner. We describe two algorithms, each requiring $O(N + V + E)$ time. The first requires $O(V^2E)$ space and the second requires $O(VE)$ space. The first algorithm has a more general methodology and may be more useful in certain applications. It is used in the weighted case in this paper.

An $O(N \log V + VE)$ algorithm for sorting the nodes of the computation tree in increasing order of weight is also presented here and is based on the fact that there are a bounded number of exchanges that change one spanning tree into another. To output the spanning trees in sorted order, however, requires $O(NV)$ operations. The scheme presented betters the $O(N \log N)$ time heapsort used by Gabow [Ga77], which results in a total time complexity of $O(NE + N \log N)$.

In a companion paper [KR92], we use similar techniques to enumerate all spanning trees of a directed graph in $O(NV)$ time, improving upon the previous best known bound of $O(NE)$ time [GM78].

Section 2 describes the generation of spanning trees in undirected and unweighted graphs and §3 describes a way of ordering the spanning trees in the computation tree for weighted graphs. Each of the sections contains a description of the algorithms and proofs of their correctness and complexity.

2. Undirected spanning tree enumeration. Let G be an undirected graph with V vertices, E edges, and N spanning trees. $E(G)$ refers to the set of edges of the graph G.

2.1. Algorithm outline. In this section, we present an outline of the algorithm for generating all spanning trees of an undirected graph.

The algorithm starts off with a spanning tree T, and generates all other spanning trees from T by replacing edges in T by edges outside T. For undirected graphs, an edge in a fundamental cycle of the graph can be replaced by its corresponding nontree edge to result in a new spanning tree. Thus, a number of spanning trees can be generated from a single spanning tree by exchanging edges in a fundamental cycle with the corresponding nontree edge. This computation can be represented by a computation tree with spanning tree T at its root and the spanning trees resulting from these exchanges at its sons. To generate other spanning trees, these sons are expanded recursively in the same manner as the root. Thus each node in the computation tree is associated with a spanning tree of G.

We need to ensure that each spanning tree is generated exactly once. This is done by a search tree-type computation tree which uses the inclusion/exclusion principle. To aid the construction of the computation tree, at every node in the computation tree two sets with the following classification are maintained. For a node x in the computation tree, the set IN_x consists of edges which are always included in all spanning trees at x and its descendants in the computation tree. The set OUT_x contains edges which are not included in any spanning tree at x or at its descendants in the computation tree. We let S_x denote the spanning tree generated at x and G_x denote the current graph obtained by contracting edges in IN_x and removing edges in OUT_x. Note that G_x may be a multigraph. We define $CYCLE_x$ to be the set of fundamental cycles of nontree edges (with respect to S_x) which are in G_x. We now formally define the computation tree $C(G)$ with respect to the graph G. $C(G)$ has a spanning tree of G associated with every node. The computation tree starts with an arbitrary spanning tree at the root. Let A be a node in the computation tree and let S_A be the spanning tree associated with A. Let f be an edge not in OUT_A or S_A and let $c_f = (e_1, e_2, \ldots, e_k)$ be the fundamental cycle

in G_A formed by f with respect to S_A. Then A has as its sons B_i, $1 \leq i \leq k+1$. (See Fig. 1; each edge is labeled with the pair of edges exchanged.) For $1 \leq i \leq k$, B_i corresponds to the spanning tree obtained by the exchange (e_i, f). Note that e_i is not already in IN_A because edges in IN_A are contracted in G_A. B_{k+1} corresponds to a node in the computation tree such that no descendant of the node has f in the spanning trees generated. Note that the tree at B_{k+1} is the same as the one at its parent.

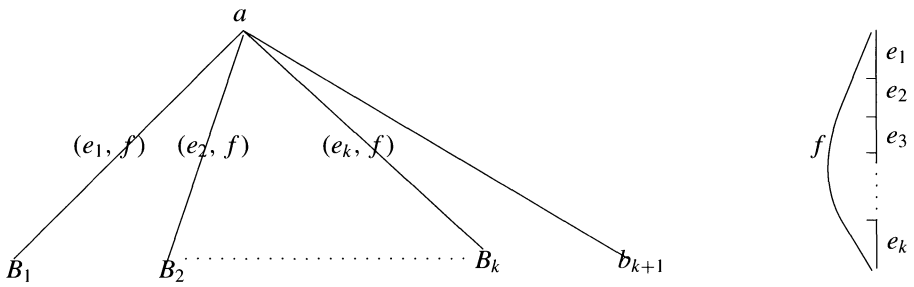


FIG. 1. $C(G)$.

The IN and OUT sets are formally defined as follows.

IN_{B_j} $= IN_A \cup \{e_1, e_2, \ldots, e_{j-1}\} \cup \{f\}$, for $1 \leq j \leq k$

OUT_{B_j} $= OUT_A \cup \{e_j\}$, for $1 \leq j \leq k$

$IN_{B_{k+1}}$ $= IN_A$

$OUT_{B_{k+1}}$ $= OUT_A \cup \{f\}$

S_{B_j} $= S_A - \{e_j\} \cup \{f\}$, for $1 \leq j \leq k$

$E(G_{B_j})$ $= E(G_A) - \{e_j\}$ *with edges* $e_1, e_2, \ldots, e_{j-1}$, f *contracted*, for $1 \leq j \leq k$

$CYCLE_{B_j}$ $= CYCLE_A$ **com** c_f *with edges* $e_1, e_2, \ldots, e_{j-1}$ *contracted in each resultant cycle* , for $1 \leq j \leq k$
{Here the **com** operation combines (see Fig. 3) all cycles in $CYCLE_A$ which contain edge e_j with c_f and leaves the other cycles intact}

$S_{B_{k+1}}$ $= S_A$

$E(G_{B_{k+1}})$ $= E(G_A) - \{f\}$

$CYCLE_{B_{k+1}}$ $= CYCLE_A - \{c_f\}$

The IN and OUT sets for the root x of the computation tree are both empty. S_x is any spanning tree, G_x is the original graph G, and $CYCLE_x$ is the set of fundamental cycles in G with respect to S. Before we show how to generate the computation tree, we show that $C(G)$ suffices to generate all the spanning trees of G.

LEMMA 2.1. *The computation tree has at its internal nodes and leaves all the spanning trees of* G.

Proof. The proof follows from induction and the inclusion/exclusion principle. The inclusion/exclusion is implemented in the computation tree as follows: Let A be the root node of the computation tree. The subtrees rooted at B_1 through B_k together form the computation tree of the spanning trees, which have the edge f in them. B_{k+1} is the root of the computation tree which computes all spanning trees not containing the edge f. Within the set of spanning trees which contain f, the subtree rooted at B_1 generates spanning trees without e_1, whereas

$B_2, \ldots B_k$ generate subtrees with e_1. A similar inclusion/exclusion process is repeated at each of the nodes $B_2, \ldots B_k$, i.e., the computation subtree rooted at B_j corresponds to the set of spanning trees which contain the edges $e_1, \ldots, e_{j-1}$ but not e_j. Moreover, the computation subtree corresponding to the inclusion of $e_1, \ldots, e_k$ and f need not be included in the computation tree since this choice of edges forms a cycle and will not lead to any spanning tree. □

In order to analyze the algorithms, it is convenient to define a compressed form $C'(G)$ of $C(G)$. Having generated node A with spanning tree S_A, we find the fundamental cycle corresponding to some nontree edge $f \in G_A$ and then the sons $B_1, \ldots, B_{k+1}$ according to the above description are generated. However, note that the sons of B_{k+1} will be obtained by using another nontree edge relative to S_A in a graph where f is absent. This is repeated along the entire rightmost branch of the computation tree. Since all these computations are with respect to S_A alone, we can obtain a compressed version of the computation tree called $C'(G)$ by considering all the fundamental cycles at node A and applying the inclusion/exclusion principle over the nontree edges. Figure 2 illustrates the compression. The compressed computation tree $C'(G)$ has the advantage that each node in the tree corresponds to a unique spanning tree. Since the compression does not eliminate nodes with distinct spanning trees in $C(G)$, $C'(G)$ generates all spanning trees of G. Each node of $C(G)$ is associated with exactly one node of $C'(G)$ and we refer to both nodes by the same name.

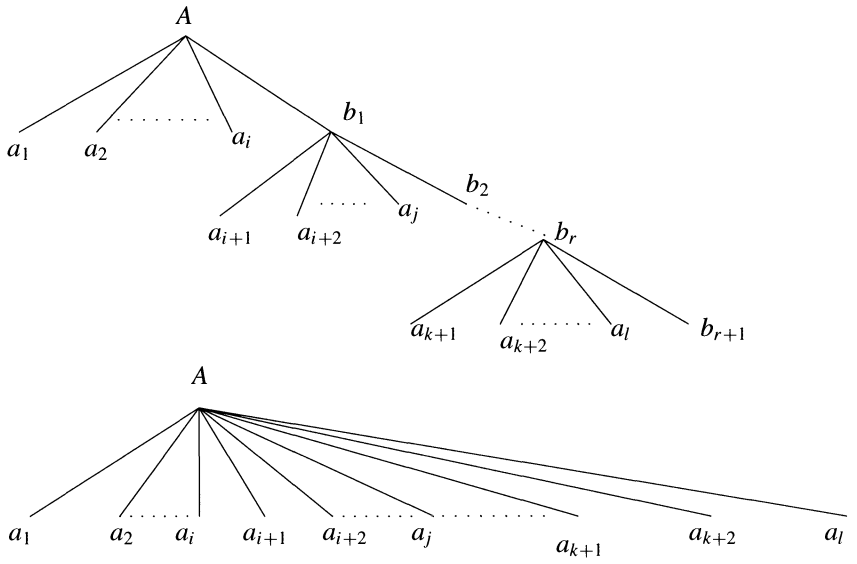


FIG. 2. $C'(G)$.

To achieve the construction of $C(G)$ in linear time, we outline schemes for finding the fundamental cycles and generating the sons of a node. An important issue in generating $C(G)$ is the computation of the set of fundamental cycles at each node of $C(G)$. Note that the current tree T has been obtained by replacing an edge e in the previous tree T' by a nontree edge f. This affects all the fundamental cycles containing the edge e. Each fundamental cycle containing edge e must now be *combined* with the fundamental cycle of f, i.e., a new cycle is obtained by removing edges common to both cycles (see Fig. 3).

In Algorithm 1, at each node A in $C(G)$, an arbitrary nontree edge f in G_A is chosen for exchanging. After each exchange is performed, each of the fundamental cycles affected

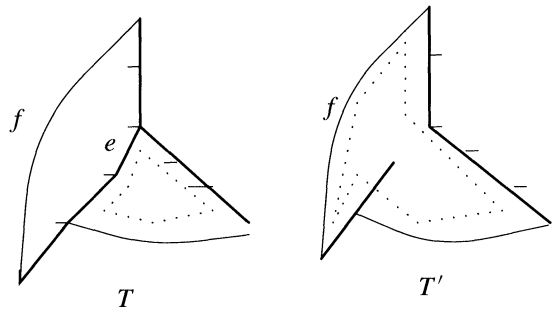


FIG. 3. *Cycle Combination*

by that exchange is combined with the fundamental cycle of f. This is done by scanning the affected cycles and changing them in time proportional to the sum of the lengths of the *resulting* cycles. Each fundamental cycle generates a number of spanning trees equal to the number of tree edges in it by exchanging with the corresponding nontree edge. This ensures that the computation tree is generated in linear time. This approach requires $O(V^2E)$ space and the data structure for maintaining the fundamental cycles is slightly elaborate. Furthermore, this involves repeated scanning of an edge which occurs in more than one cycle.

To reduce space and to simplify and speed up the data structures, we describe Algorithm 2. In this algorithm, the tree S_A is always a depth-first search tree (d.f.s tree) of G_A at each node A of $C(G)$. This is ensured in the following manner. We start at the root of $C(G)$ with the d.f.s tree of G. Further, at each node A of $C(G)$, the nontree edges in G_A are considered for exchanges in a particular order. The order is given by the increasing post-order number of the upper (i.e., closer to the root) endpoints of the nontree edges. Note that since S_A is a d.f.s tree of G_A, all the nontree edges are back edges with respect to S_A. Clearly, finding the fundamental cycle of a nontree edge f is now straightforward: it simply involves marching up S_A from the lower endpoint of f to its upper endpoint. As we shall describe later, combining fundamental cycles now becomes a matter of simply changing the endpoints of some edges. The total space used in this scheme is $O(VE)$.

We remark that while Algorithm 2 is more efficient, Algorithm 1 is more general. In particular, it can be used as the base scheme in generating the spanning trees of a weighted graph in order (§4) while Algorithm 2 fails in that application because of its depth-first restriction.

2.2. Algorithm 1 description. The main algorithm, *Main*, has as input a graph G. It generates a spanning tree T of G corresponding to the root of the computation tree and computes the fundamental cycles with respect to T. *Main* also initializes the data structures, which will be described shortly.

ALGO Main(G);
Find a spanning tree, T, of G;
Initialize data structures;
Gen(T);
End Main.

The heart of the algorithm is the generation scheme *Gen*, which generates the sons of a node in the computation tree and recursively generates the subtrees rooted at them. The entire computation tree is thus generated in an pre-order traversal of the tree. The following is an outline of the scheme.

Gen picks an edge f from F, the data structure which stores all nontree multiedges in the current graph. It then determines the fundamental cycle $c_f = (e_1, \ldots, e_k)$ of f with respect to T, the current spanning tree. Note that $e_1, \ldots, e_k$ are only the tree edges in c_f, with the actual fundamental cycle formed by these edges along with the edge f.

Each edge in c_f in chosen for exchange with f in turn. When a tree edge e_i is chosen for exchange, it is removed from the current graph (i.e., put in the OUT set). As a result, all fundamental cycles containing this edge are now modified, i.e., they now have to be combined with c_f. Note that by this point, the edges $e_1, \ldots, e_{i-1}$ are already added to the IN set and thus are already contracted in the graph and in c_f. Furthermore, the edge f must be contracted in the current graph as it is added to the IN set. These changes are made by procedure *Prepare–for–son*.

After the computation subtree rooted at the node corresponding to the exchange (e_i, f) is constructed recursively, the edge e_i must now be contracted in the current graph (i.e., added to the IN set). This is done in the procedure *Prepare–for–sons'–sibling–branch*.

Finally, the last branch of $C(G)$ involves removal of the edge f from the current graph; this is done in procedure *Prepare–for–final–son*. Note that before returning from *Gen*, the state of the data structures is restored to that at the time its invocation.

The output of *Gen* is derived from the variable $CHANGES$. $CHANGES$ accumulates the exchanges used to derive the current spanning tree from the previous spanning tree generated. $CHANGES$ is initialized to ϕ by $Main$ and reset whenever a spanning tree is output. It is modified whenever an edge of $C(G)$ is traversed in the downward direction and this modification is reversed while backtracking upwards along this edge.

ALGO Gen(T);
$mf \leftarrow$ A multiedge in F;
$f \leftarrow$ An edge in mf;
$F \leftarrow F - mf$;
$c_f \leftarrow$ the fundamental cycle of f w.r.t T in G;
/* $c_f \leftarrow (e_1, e_2, \ldots, e_k)$ in the order they occur in the cycle ; */
For $i = 1$ to k **do**
$T' \leftarrow T + f - e_i$;
$CHANGES \leftarrow CHANGES + \{(e_i, f)\}$;
Output $CHANGES$;
/*These are the difference from the last tree generated; */
$CHANGES \leftarrow \phi$;
Prepare–for–son(e_i);
If $F \neq \phi$ **then** *Gen(T)*;
$CHANGES \leftarrow CHANGES + \{(f, e_i)\}$;
If $i < k$ **then** *Prepare–for–sons'–sibling–branch*(e_i);
End For
Restore all changes made to the graph and the data
structures in the above *For* loop;
Restore multiedge $mf - f$ to F;
Prepare–for–final–son( );
If $F \neq \phi$ **then** *Gen*(T') ;
Restore changes made to the graph and the data
structures in *Prepare–for–final–son*;
Restore edge f to multiedge mf in F;
End Gen.

Next, we describe the subprocedures used by Gen. Their description and performance is linked to the data structures used. Clearly, the following data structures suffice.

1. The list F of nontree multiedges.
2. A data structure AG to maintain the current graph, which supports the operations of contracting and deleting edges. As described later, we store only nontree edges in AG. The tree edges are stored in the following cycle data structure.
3. A data structure C that stores all fundamental cycles of the current graph, which allows for determining the fundamental cycle of a particular nontree edge, determining all fundamental cycles which contain a particular tree edge, and combining all cycles containing a particular tree edge with a given cycle.

We describe the subprocedures used by *Gen* in terms of the operations performed upon the data structures mentioned above. Each operation will be described in detail later.

PROCEDURE Prepare–for–son(e_i);
    If $i = 1$ **then** contract nontree edge f in AG;
    Combine all cycles in C which contain tree edge
      e_i with cycle in C corresponding to edge f;
End Prepare–for–son.

PROCEDURE Prepare–for–sons'–sibling-branch(e_i);
    Contract e_i in all cycles in C containing e_i;
    Modify AG in order to reflect the contraction of e_i;
End Prepare–for–sons'–sibling-branch.

PROCEDURE Prepare–for–final–son;
    Delete nontree edge f from AG;
    Modify C to reflect the deletion of nontree edge f;
End Prepare–for–final–son.

Data structures. Next, we give a high level description of these data structures and the operations performed upon them by the subprocedures. As we will show later, the time spent in each of these operations can be amortized to nodes of $C'(G)$ and $C(G)$ in a manner such that each node gets charged a constant amount.

The graph data structure. The graph (which is a multigraph, in general) is maintained as an adjacency list structure AG of multiedges in the usual manner with just the following difference: edges that are not in the current spanning tree are maintained. Note that edges constituting a multiedge are clubbed together in this structure. The edges in the current spanning tree are maintained as part of the data structure storing the fundamental cycles.

Operations on the graph. Contraction and deletion of edges in AG is done as follows. Deleting a nontree edge from AG is straightforward and takes constant time, given a pointer to that edge. Next, consider the contraction of edges. Note that only tree edges are contracted in the algorithm. Contraction of a tree edge involves merging the adjacency lists of the two endpoints of the edge and takes time proportional to the number of multiedges in the two adjacency lists. While this merger is performed, one of the multiedges may become a self-loop; this multiedge is removed. This ensures that each nontree edge in AG has a fundamental cycle with at least one tree edge at all times.

LEMMA 2.2. *Data structure AG allows deletion and contraction of nontree edges in constant time and contraction of a tree edge in time proportional to the number of multiedges incident upon the two endpoints of that edge.*

The data structure *F*. The list F is maintained in the obvious way as a list of lists, with each list storing a nontree multiedge. These multiedges are linked to the correspond-

ing multiedges in AG, so changes in AG can be reflected in F in the same time bounds.

The cycle data structure. The data structure C for storing fundamental cycles is as follows. The tree edges in each fundamental cycle are stored in a circular doubly linked list in the order in which they appear. This list is accessible by the corresponding nontree multiedge. Note that all edges which constitute a multiedge share the same fundamental cycle. For each tree edge e which is not in the IN set, to find all fundamental cycles containing e, we maintain a list of fundamental cycles containing e. Furthermore, given a tree edge e which is not in the IN set, we need to be able to delete e from all cycles containing e in time proportional to the number of these cycles. Therefore, for each tree edge e, the list containing fundamental cycles also stores pointers to the location of e in each of the cycles.

Operations on the cycles. Consider deletion of nontree edge f first. If f is not part of a multiedge then its fundamental cycle must be removed from C and C must then be modified appropriately. This is easily accomplished in time proportional to the size of this fundamental cycle. If f is part of a multiedge, then no changes need be made to C. Next, consider contractions of tree edges. When a tree edge e_i is contracted, the change must be reflected in each of the cycles containing e_i. This takes time proportional to the number of such cycles.

It remains to describe the operation of combining cycles. Consider the generation of son B_i of A. All fundamental cycles containing e_i in the current graph must be combined with c'_f, the fundamental cycle of f with respect to S_A with the edges $e_1, \ldots, e_{i-1}$ contracted.

The cycle combination algorithm. We show how to combine cycles c and c_f in time proportional to the size of the resulting cycle. Note that the resulting cycle does not have any tree edges common to both c and c_f. Therefore, the main aim of the combining operation is to combine c and c_f while avoiding the chain of edges common to both c and c_f. This is done as follows. Let a_1 and a_2 be the endpoints of the nontree edge g associated with fundamental cycle c. Let a_3 and a_4 be the endpoints of the nontree edge f associated with fundamental cycle c_f. We show how to determine the endpoints of the chain D of tree edges, which constitutes the portion common to c and c_f in time proportional to $|c| + |c_f| - |D|$. Clearly, by knowing these two endpoints, the resulting cycle can easily be obtained in same time bound. Thus the time required to combine two cycles is proportional to the size of the resulting cycle.

We traverse c and c_f by using four pointers $p_1 \ldots p_4$, two per cycle, with p_i pointing to a_i initially. There are a number of rounds; in each round, each pointer traverses one edge of the cycle moving towards the other end of the cycle. Pointer p_i stops moving when either it reaches an edge which has been traversed previously (by some pointer) or it meets another pointer p_j. The latter conditions holds when either p_i and p_j point to the same vertex or they cross each other while traversing an edge. The procedure stops when all four pointers have stopped moving. The number of rounds is at most $\max\{|c|, |c_f|\} - |D|$ because as long as the above procedure continues, at least one of the four pointers must be outside D.

We show that the endpoints of D can be inferred from the final positions of the four pointers. Clearly, some two pointers must meet each other during the above procedure. There are three cases to consider depending upon which pointers meet. (See Fig. 4; pointers are shown in their final positions in each case.) First, suppose that the pointers which traverse c meet each other. Then the pointers which traverse c_f must be finally located at the two endpoints of D. The case where the two pointers which traverse c_f meet can be handled similarly. Second, suppose a pointer which traverses c meets a pointer which traverses c_f, but the remaining two pointers do not meet. These two remaining pointers must be finally located at the endpoints of D. Third, suppose each pointer which traverses c meets at least one pointer which traverses c_f. Then, it can easily be verified that each pointer which traverses c meets exactly one of the pointers traversing c_f. In this case, two of the pointers, one which traverses

c and one which traverses c_f, must be finally located at one endpoint of D while the other two pointers must be located at the other endpoint.

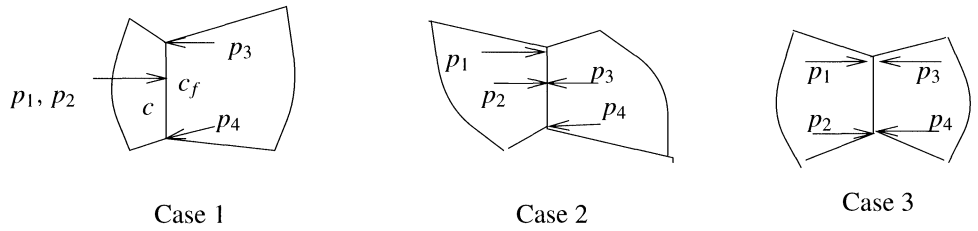


FIG. 4. *Three cases in the cycle combination algorithm.*

It follows that the cycle combination can be achieved in time proportional to the number of edges in the resulting cycle.

LEMMA 2.3. *The data structure C allows for the following items.*

1. *Deletion of nontree edge f in time proportional to the size of its fundamental cycle.*
2. *Contraction of tree edge e_i in time proportional to the number of cycle it is contained in.*
3. *Combining two cycles in time proportional to the size of the resulting cycle.*

This ends the description of the data structures.

2.2.1. Analysis. We start with the analysis of the time complexity of Algorithm 1. First, we show that the total output size of the algorithm is $O(N)$.

LEMMA 2.4. *The number of exchanges output by Gen is at most* $2N$.

Proof. Consider internal node x of $C(G)$. For each son y of x, except the last, one exchange is added to $CHANGES$ when y is generated and its opposite exchange ((e, f) is the exchange opposite to (f, e)) is added to $CHANGES$ after the sub-tree rooted at y has been generated. Therefore, the number of exchanges added to $CHANGES$ is at most $2N$. Furthermore, every time $CHANGES$ is output, it is reset to ϕ immediately. The lemma follows. □

The following lemma is the key one in obtaining the final complexity.

LEMMA 2.5. *The work done at each node A of $C(G)$ is $O(|s(A)|+|g(A)|+r(A))$, where $s(A)$ is the set of sons of A in $C(G)$, $g(A)$ is the set of sons in $C'(G)$ of nodes in $s(A)$, and $r(A)$ is the number of exchanges output at node A.*

Proof. The work done to output exchanges at A is at most $O(r(A))$. We consider the other operations performed at node A and show how to amortize the time spent in these operations to nodes in $s(A)$ and $g(A)$

Clearly, the operations performed in *Gen* (excluding those performed in the subprocedures) take time proportional to the number of sons of A. We look at the operations performed next in the subprocedures.

First, consider *Prepare–for–son*(e_i). Through Lemma 2.2, contracting f takes constant time. According to Lemma 2.3, the time to combine all cycles containing e_i with the fundamental cycle for f takes time proportional to the sum sizes of the resulting cycles. Each resulting cycle leads to a number of sons of B_i in $C'(G)$ equal to its size. Thus the time spent in combining cycles can be charged to the sons of node B_i in $C'(G)$. Consequently, each node in $g(A)$ gets charged once for cycle combinations over all calls to *Prepare–for–son*.

Next, consider *Prepare–for–sons'–sibling–branch*(e_i). First, consider contraction of the edge e_i in C. By Lemma 2.3, contracting e_i in C takes time proportional to the number of cycles it is contained in. Consider each of these cycles following the contraction of e_i.

At most one of these, c_j, say, will not contain a tree edge, as edges corresponding to a multiedge are clubbed together and self-loops are removed as they are formed. Therefore, each of these cycles except c_j and c'_f, the fundamental cycle of f with respect to S_A with the edges $e_1, \ldots, e_{i-1}, e_i$ contracted, will lead to at least one son of B_{i+1} in $C'(G)$. Next, consider the contraction of edge e_i in AG. By Lemma 2.2, this takes time proportional to the number of multiedges incident upon the two endpoints of e_i. Consider the set M of these multiedges, excluding the one multiedge which is converted to a self-loop upon contraction, if any. Following the contraction, we see that each multiedge $g \in M$ with tree edges other than the one contracted has a fundamental cycle containing at least one tree edge. (Recall that self-loops are removed as they are formed.) Therefore, every edge in each such multiedge g, with the possible exception of the multiedge having the same endpoints as f, gives rise to at least one son of B_{i+1} in $C'(G)$. Thus, the time spent in *Prepare–for–sons'–sibling–branch*(e_i) can be charged to the sons of node B_{i+1} in $C'(G)$. Therefore, each node in $g(A)$ gets charged at most twice over all calls to *Prepare–for–sons'–sibling–branch*.

Finally, consider *Prepare–for–final–son*. By Lemma 2.2, deletion of f from AG takes constant time. By Lemma 2.3, deletion of f from C takes time proportional to the size of its fundamental cycle, which, in turn, equals the number of sons of A in $C(G)$. The lemma follows. □

Next, we obtain the time complexity of *Gen* and *Main*.

THEOREM 2.6. *All spanning trees can be correctly generated in* $O(N + V + E)$ *time by Main.*

Proof. First, note that *Gen* correctly computes the spanning trees at sons of a node A of the computation tree. This follows from the fact that the cycles and the graph are correctly updated after the inclusion of edges into the IN and OUT sets. The correctness of the updates is evident from the operations on the data structures discussed in detail before. Also note that *Gen* correctly maintains $CHANGES$ which stores the difference between the current tree being output and the last spanning tree generated. Thus *Gen* correctly generates the computation tree and outputs the tree differences.

We next compute the time complexity. The preprocessing steps in *Main* before calling *Gen* is called require $O(V + E)$ time, in addition to the time required for setting up the cycle data structure. The latter can be charged to the sons of the root node of $C'(G)$. We show next that the time taken by the call to *Gen* in *Main* is $O(N)$.

The total time for outputting exchanges over all invocations of *Gen* is $O(N)$ by Lemma 2.4. Next, consider the time spent in a particular invocation of *Gen*, minus the time for outputting exchanges in that invocation. Let this invocation correspond to the creation of the sons of node x in $C(G)$. By Lemma 2.5, the time taken by this invocation of *Gen* is at most $O(|s(x)|+|g(x)|)$. Summing over all nodes of $C(G)$, it follows that the time taken by the call to *Gen* in *Main* is $O(N)$. The lemma follows. □

Next, we analyze the space complexity.

THEOREM 2.7. *The space required by the spanning tree enumeration algorithm, Main, is* $O(V^2E)$.

Proof. At each node of $C(G)$, changes to the data structures need to be stored to enable restoration later. These changes take $O(VE)$ space and this dominates the space requirement. The space taken to store changes when the last son of any node in $C(G)$ is generated is $O(1)$. (Only a nontree edge is deleted.) On any root-to-leaf path in $C(G)$, the number of nodes which are not the rightmost sons of their respective parents (note that this number is bounded by the height of $C'(G)$) is at most V. This is because any two spanning trees can differ in at most $V - 1$ pairs of edges. The theorem now follows from the fact that *Gen* generates $C(G)$ in a depth-first manner. □

2.3. Algorithm 2 description. We describe a second algorithm based on the use of depth-first search to construct $C(G)$.

As before, we generate the computation tree, $C(G)$ recursively: The details are as follows. At the root of the computation tree the spanning tree is constructed by a depth-first scan of the graph. In fact, we maintain the following invariant: At each node A of the computation tree, S_A is a d.f.s spanning tree of G_A. Consequently, all nontree edges in G_A are back edges with respect to S_A. This property makes the task of determining and combining fundamental cycles much easier. In particular, it is no longer necessary to maintain all the fundamental cycles in a separate data structure. Only the spanning tree S_A itself needs to be stored, with the edges in IN_A contracted. Given S_A, the fundamental cycle corresponding to a particular nontree edge $f \in G_A$ can now be found by marching along S_A from the endpoint farther from the root to the endpoint closer to the root. This is instrumental in reducing the space bound to $O(VE)$. The combination of cycles is also simplified, as we will describe below.

Moreover, in order to ensure efficiency we must ensure that all useless tree edges are not to be considered in the replacement process. Useless edges are those edges that do not give rise to an exchange and hence to a spanning tree. These are the edges that do not occur in any cycles; i.e., they are the bridges of the graph. We describe their removal below.

For convenience, let *up* and *down* denote the directions towards and away from the root, respectively. For each nontree edge, its *upper* endpoint is the one closer to the root.

2.3.1. Maintaining the d.f.s. invariant. We show how to maintain the invariant regarding the d.f.s nature of each spanning tree. Suppose node A of $C(G)$ has been generated and S_A is a d.f.s tree of G_A. Assume that the vertices of S_A are numbered by a post-order traversal of S_A. At node A, we select the nontree edge f, whose upper endpoint has the least post-order number among all the nontree edges in G_A. Edge f is used as the replacement edge. Furthermore, the tree edges in the fundamental cycle of f with respect to S_A are replaced in order, starting from the edge farthest from the root and proceeding upwards. Let $e_1, \ldots, e_k$ be these tree edges in that order, and let $e_i = (x_{i-1}, x_i)$. We claim that for each of the sons $B_1, \ldots, B_k$ of A, the spanning trees at these nodes are d.f.s trees of G_A. This is shown in Lemma 2.8. Before we prove this lemma, we describe the various operations that take place when these sons are generated. This is helpful in describing the proof of Lemma 2.8.

Let T be the data structure storing the spanning trees. T is implemented in the usual manner as a set of parent pointers and child pointers. Before any of the sons of A have been generated, T stores S_A. Consider the generation of son B_i of A. The edges $e_1, \ldots, e_{i-1}$, f are all in the IN set and therefore, must be contracted in T. As a result of this contraction, the sons of $x_0, \ldots, x_{i-1}$ in T now become sons of x_k. Furthermore, all nontree edges in G_A which have one of $x_0, \ldots, x_{i-1}$ as their lower endpoints must have their lower endpoint changed to x_k. Since edges $e_1, \ldots, e_{i-2}$, f would already have been contracted before son B_{i-1} is generated, only the sons of x_{i-1} need to be made sons of x_k, and only the lower endpoints of those nontree edges which currently have lower endpoint x_{i-1} need be set to x_k when son B_i is generated (see Fig. 5). Note that with this, we have effectively achieved the task of cycle combination.

LEMMA 2.8. *For each son B_i, $1 \leq i \leq k+1$ of A, all nontree edges in G_{B_i} are back edges with respect to S_{B_i}, assuming that all nontree edges in G_A are back edges with respect to S_A.*

Proof. For B_{k+1}, the lemma is easily seen to be true. Consider son B_i, $1 \leq i \leq k$. Consider any nontree edge g in G_{B_i}. Note that $g \in G_A$. The upper endpoint of g in S_A must be either x_k or an ancestor of x_k in S_A. From the previous paragraph, when S_{B_i} is obtained from S_A, the lower endpoint of g either remains unchanged or is changed to x_k. Furthermore, any descendant of x_k in S_A remains a descendant of x_k is S_{B_i}. Thus all nontree edges in G_{B_i} are back edges with respect to S_{B_i}. The lemma follows. □

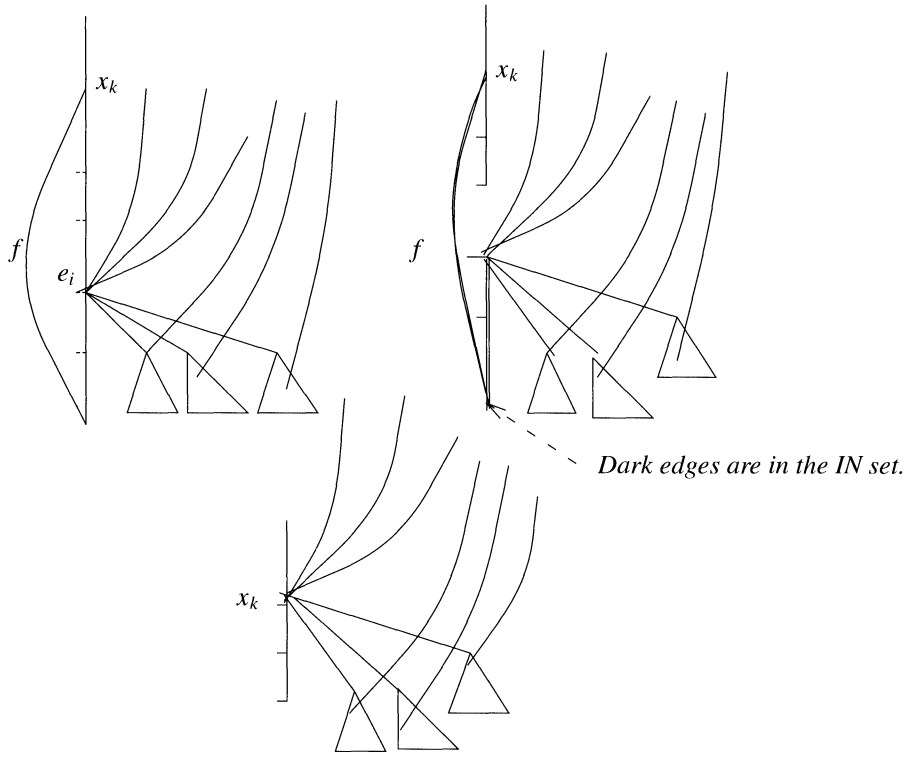


FIG. 5. *Generation of* B_i.

2.3.2. Detecting and removing bridge edges in T. Now we show how bridge edges in T are detected and removed as they are formed. This is initially accomplished by dividing the graph into its biconnected components and generating the spanning trees for each component separately. The spanning trees for the entire graph can be obtained easily from the spanning trees of its biconnected components.

Next, we show how bridges are detected at node A of $C(G)$ when its sons $B_1, \ldots, B_{k+1}$ are being generated, assuming that at A, T has no bridge edges. The following observations are key. Clearly, the only edges in T which can be converted to bridge edges are the edges $e_1, \ldots, e_k$. This can happen when either one of the e_i's is deleted or when f is deleted. The condition that characterizes the situation when e_i becomes a bridge edge is as follows. Let $j \geq i - 1$ be the smallest number such that x_j has either a branch (i.e., at least two sons) or an incident nontree edge. (Note that x_j would be the lower endpoint of such a nontree edge.) If $j \geq i$, then e_i is a bridge edge and so are the edges $e_{i+1}, \ldots, e_j$. But $e_{j+1}, \ldots, e_k$ remain nonbridge edges.

The bridge detection procedure which results from the above observations is as follows. First, consider the deletion of edge e_i. By this point, edges $e_1, \ldots, e_{i-1}$ are already contracted. Among the edges $e_{i+1}, \ldots, e_k$, those edges which are converted to bridge edges by this deletion are ascertained by traversing these edges in the above sequence until a vertex x_j, $j \geq i$, is found such that x_j has either a branch (i.e., at least two sons before the deletion of e_i) or a an incident nontree edge. The edges traversed are converted to bridge edges by the deletion of e_i. These edges are removed from T by simply removing the vertex x_{j-1} from T. Furthermore, note that when the edges $e_{i+1}, \ldots, e_j$ are deleted subsequently, bridge removal can be accomplished in

each case by the deletion of x_{j-1} from T. Later, when e_{j+1} is deleted, bridge detection will be done by traversing the edges $e_{j+2}, \ldots, e_k$ in that order. Next, consider the deletion of edge f. Bridge detection is done by traversing the edges $e_1, \ldots, e_k$ until a vertex x_j, $j \geq 0$, is found such that x_j has either a branch or an incident nontree edge. Bridge removal involves removing x_j from T.

LEMMA 2.9. *The total time taken for bridge detection over deletions of all of* $e_1, \ldots, e_k$ *is* $O(k)$. *The time taken for bridge detection when* f *is deleted is also* $O(k)$.

2.3.3. Data structures. Next, we describe details of the data structures used as well as the operations on them along with the time complexity of executing them. Later, we will show that a result similar to Lemma 2.5 holds for this algorithm too.

Storing tree edges. The data structure T stores the current spanning tree in the form of a parent pointer and a list of child pointers for each node, with only edges not currently in the IN set present. Determining the fundamental cycle of a particular nontree edge and deletion of a tree edge are trivial.

Consider edge contraction next. When edge $e_i = (x_{i-1}, x_i)$ is contracted, the sons of x_i are made sons of x_k. This takes time proportional to the number of sons of x_i. In order to account for this time, we use the invariant that each edge in T is a nonbridge edge in the current graph, i.e., it occurs in at least one fundamental cycle.

LEMMA 2.10. *Data structure* T *allows for*

1. *Determining the fundamental cycle of a nontree edge in time proportional to the length of the cycle.*
2. *Contracting tree edge* e_i *in time proportional to the number of sons of vertex* x_i *in* T.

Storing nontree edges. Recall that nontree edges have to be ordered by their upper endpoints. Furthermore, given a vertex v, we need to be able to determine all those nontree edges which have a lower endpoint v. These edges are required during the contraction of edges.

The following lemma is important for maintaining nontree edges in the requisite order. It shows that this order is independent of the particular node of $C(G)$ that is considered. Therefore, we can initially order nontree edges in an order given by a post-order traversal of the spanning tree at the root of $C(G)$; this order will hold throughout the algorithm.

LEMMA 2.11. *Suppose for edges* $e_a, e_b \in G_{B_i}$, *the upper endpoint of* e_a *appears before the upper endpoint of* e_b *in the post-order ordering of vertices in* S_{B_i}, $1 \leq i \leq k+1$. *Then the upper endpoint of* e_a *appears before the upper endpoint of* e_b *in the post-order ordering of vertices in* S_A.

Proof. This is clearly true for $i = k+1$. So suppose $i < k+1$. Edge $f = (x_0, x_k)$ has the least upper endpoint in the post-order ordering of all vertices in S_A. When B_i is generated, only the portion of S_A consisting of descendants of x_k is modified to give S_{B_i}. No descendants of x_k can have a greater post-order number than x_k. The lemma follows. □

As before, the data structure for storing nontree edges actually stores multiedges. This data structure has two components, *REP-LIST* and *ADJ-LIST*. Modifications to one of them can be reflected in the other without any extra time overhead by keeping pointers between the corresponding multiedges in the two structures.

REP-LIST is simply a list of nontree multiedges eligible for replacement, with the multiedges appearing in the requisite order. At each node A of $C(G)$, an edge f from the first multiedge in this list is picked for replacement and this edge is deleted from this list to generate the last son of A; selecting the replacement edge and deleting an edge thus takes constant time.

For each vertex v, *ADJ-LIST* stores a list of nontree multiedges which have v for their lower endpoint. Clearly, for vertex x_i, the lower endpoints of all edges which currently have

lower endpoint x_i can be changed to x_k by merging the multiedge list for x_i with that for x_k. Any self-loop formed is removed. (Note that since edges are organized into multiedges, at most one self-loop is formed per merger.) This can be done in time proportional to the number of multiedges in the two lists.

LEMMA 2.12. *Data structures* REP-LIST *and* ADJ-LIST *can be maintained such that*

1. *The replacement edge can be selected and deleted in constant time.*
2. *The lower endpoints of all nontree edges with lower endpoint x_i can be transferred to x_k in time proportional to the number of nontree multiedges incident upon the two vertices.*
3. *The ordering of nontree edges remains unchanged.*

2.3.4. Algorithm 2 pseudocode. For detail and clarity, we present the pseudocode of Algorithm 2. The main procedure *Main*2 sets up the initial data structures and then uses *Gen*2 to generate all spanning trees of G. The basic framework of *Main*2 and *Gen*2 is similar to that of *Main* and *Gen* but with one notable difference. *Main2* splits G into its biconnected components and generates all spanning trees of G by using *Gen*2 to generate all spanning trees of each biconnected component. *Gen*2 uses the procedures *Combine-Cycles*, *Remove-Bridges*1, and *Remove-Bridges*2. The first of these makes the changes required when edge edge e_{i-1} is contraction and edge e_i is deleted in order to generate son B_i of A. The last two perform the detection and removal of bridges from the graph resulting from deletion of edges $e_1, \ldots, e_k$ and the deletion of edge f, respectively. A stack *STACK* is used to store changes made to data structures before recursing so as to undo these changes after the recursion completes.

ALGO Main2(G)

Determine biconnected components of $G = (G_1, G_2, \ldots, G_k)$;
$T_i \leftarrow$ D.F.S tree of G_i, $1 \leq i \leq k$;
For each i, $1 \leq i \leq k$ do
    Compute *REP-LIST* and *ADJ-LIST*;
Use *Gen*2 to output all spanning trees of G in the
lexicographic ordering of the biconnected components.
End Main2.

ALGO Gen2(T)

If *REP-LIST*$= \phi$ **then** return;
$f = (u, v) \leftarrow$ First edge in *REP-LIST*;
    /* u represents the lower endpoint of the
    back edge in the DFS tree */
$c_f \leftarrow (x_1 = u, x_2, \ldots, x_{k+1} = v)$;
    /*the sequence of vertices in the fundamental cycle
    of f w.r.t T from bottom upwards */
REP-LIST $\leftarrow$ *REP-LIST* $- f$;
LAST $\leftarrow$ The first ancestor of x_1 whose
    father either has a branch or a back edge;
      /* Useful for bridge elimination, $LAST$ is local to *Gen*2 */
For $i = 1$ to k **do**
    $e \leftarrow (x_i, x_{i+1})$;
    $LAST \leftarrow$ *Remove-bridges*1$(LAST, x_i)$;
    $T \leftarrow T - e \cup f$;
    $IN \leftarrow IN + \{f\}$; /* Update the IN set*/

```
        OUT ← OUT + {e}; /* Update the OUT set*/
        CHANGES ← CHANGES +{(e, f)};
        Output CHANGES;
              /*Output differences from the last tree generated */
        CHANGES ← φ;
        If i < k Combine-cycles(i, x_k, f);
        Gen2(T);
        Restore LAST to T;
        CHANGES← CHANGES +{(f, e)};
        IN ← IN − {f};
        OUT ← OUT − {e};
        T ← T − f + e; /* restoring T */
        IN ← IN + {e}; /* adding edge removed to OUT set */
        inlist ← inlist ∪ e;
    End For;
    IN ← IN − inlist;
    OUT ← OUT + {f};
    Restore-changes;
        /* Undo all changes made by all calls to COMBINE-CYCLES
          in the above for loop using STACK to retrieve changes */
    ADJ-LIST[x1] ← ADJ-LIST[x1] − f;
        /* Prepares for last recursion associated with c_f */
    LAST ← Remove-bridges2( );
    Gen2(T);
    Restore vertex LAST to T;
    Restore f to the front of REP-LIST and to ADJ-LIST [x1];
    OUT ← OUT − {f};
End Gen2.

Procedure Combine-cycles(i, x_k, f);
    For each son y of x_i make father(y) ← x_k;
    Store the changes made above on STACK;
    For each sublist, l, of multiple back edges in ADJ-LIST(x_i) do
        If edges in l have upper endpoint v then
            Remove l from REP-LIST and ADJ − LIST[x_i];
                  /*Edges in l are now loops and hence removed */
            Store the changes made above on STACK;
        else
            Transfer l to ADJ-LIST(v);
            Store the changes made above on STACK;
End Combine-cycle;

Procedure Remove-bridges1(LAST, x_i);
    If x_i = father of vertex LAST in T then
        LAST ← The first (i.e., closest to x_i) ancestor of x_i in c_f
          whose father has a back edge or a branch;
            /* LAST is x_k if no such vertex exists */
    Remove vertex LAST from T;
    Return(LAST);
End Remove-bridges1.
```

Procedure Remove-bridges2( );

If $ADJ - LIST[x_1]$ is empty and x_1 has no son in T then

$LAST \leftarrow$The first (i.e., closest to x_1) ancestor of x_1 in c_f whose father has a back edge or a branch;

else

$LAST \leftarrow \phi$;

Remove vertex $LAST$ from T;

Return($LAST$);

End Remove-bridges2.

2.3.5. Time and space complexity. Lemma 2.4 holds for this algorithm too. Next, we show that Lemma 2.5 holds too.

LEMMA 2.13. *The work done at each node A of $C(G)$ is $O(|s(A)| + |g(A)| + r(A))$, where $s(A)$ is the set of sons of A in $C(G)$, $g(A)$ is the set of sons in $C'(G)$ of nodes in $s(A)$, and $r(A)$ is the number of exchanges output at node A.*

Proof. The work done to output exchanges at A is $O(r(A))$ We consider the other operations performed at node A and show how to amortize the time spent in these operations to nodes in $s(A)$ and $g(A)$.

Clearly, the operations performed in *Gen*2 (excluding those performed in the subprocedures) take time proportional to the number of sons of A. We look at the operations performed in the subprocedures next.

First, consider all invocations of procedure *Remove-Bridges*1 and *Remove-Bridges*2. By Lemma 2.9, the time spent in these procedures is proportional to the number of sons of A in $C(G)$.

Second, consider the call to *Combine-Cycle*(i, x_k, f). The two major operations done in this procedure are changing the parent pointers of sons of x_i to x_k and changing the lower endpoints of some nontree edges from x_i to x_k. We consider each of these in turn.

First, consider the operation of changing the parent pointers of the sons of x_i. By Lemma 2.10, this takes time proportional to the number of sons of x_i in T. Since bridges are removed from T as they are formed, for each son y of x_i in T (note that x_{i-1} is not in T currently as the edge (x_{i-1}, x_i) is in the IN set), there exists at least one nontree edge f' such that the edge (x_i, y) belongs to the fundamental cycle of f'. The exchange of f' for (x_i, y) leads to a son of B_{i+1} in $C'(G)$ to which the time for changing the parent pointer of y can be charged. Thus, over all calls to *Combine-Cycle*, each node in $g(A)$ gets charged at most once for changing parent pointers.

Second, consider the operation of changing the lower endpoints of some nontree edges from x_i to x_k. By Lemma 2.12, this takes time proportional to the number of nontree multiedges incident upon x_i and x_k. At most one of the multiedges incident upon x_i, i.e., the multiedge with upper endpoint x_k, is converted to a self-loop in the above process. All other multiedges which are incident upon any of the two vertices have a fundamental cycle containing at least one tree edge; each such fundamental cycle leads to at least one son of B_{i+1} in $C'(G)$. Therefore, the time spent in modifying lower endpoints in *Combine-Cycle*(i, x_k, f) can be charged to the grandsons of A through B_{i+1} in $C'(G)$. Thus, over all calls to *Combine-Cycle*, each node in $g(A)$ gets charged at most once for changing lower endpoints.

The lemma follows. □

THEOREM 2.14. *Algorithm* 2 *generates all spanning trees of T is $O(N + V + E)$ time correctly.*

Proof. First we consider correctness. From the discussions in the previous sections, it is easy to see that *Gen* generates the computation tree such that S_A is a d.f.s. tree of G_A. Thus the cycle and exchange generation process is correct. Note that *Gen* maintains G_A as a biconnected graph. Since *Main* calls *Gen* for each biconnected component all spanning trees can be generated by taking all possible combinations in the individual components.

Next we consider the time complexity. Without loss of generality assume that G is biconnected. If this is not the case, then for the ith biconnected component of G having N_i spanning trees, V_i vertices and E_i edges, the time taken will be $O(N_1 + V_1 + E_1)$ and the time to put the computation trees of the various components together in the obvious manner will be $O(\Pi_i N_i + \Sigma_i(V_i + E_i)) = O(N + V + E)$.

From Lemma 2.13, each node of $C(G)$ and $C'(G)$ is charged a constant amount for work done over all nodes of $C(G)$ and by Lemma 2.4, the total work done to output exchanges is $O(N)$. *Main* takes $O(V + E)$ time to construct the initial data structures. The theorem follows. □

THEOREM 2.15. *Algorithm* 2 *takes* $O(VE)$ *space.*

Proof. The space required for *REP-LIST* and *ADJ-LIST* and T is $O(V + E)$. Stack space for storing changes to be undone after returning from a recursive call is shown to be $O(VE + V^2)$ as follows. The parent vertex of any vertex is changed at most $O(V)$ times along any path of $C(G)$. The lower endpoint for any nontree edge is changed at most V times along any path in $C(G)$. An edge is deleted at most once along any path in $C(G)$. Consequently storing changes to *ADJ-LIST*, *REP-LIST* and T require $O(VE)$ space. So the total space is $O(VE)$. □

3. Generating the computation tree in increasing weight order. Next, assuming that the edges of G are weighted, we present an algorithm to generate the nodes of $C(G)$ in increasing order of weight.

The algorithm follows a branch and bound strategy on the computation tree. The root of the computation tree is now associated with the minimum spanning tree (MST) of the graph. The sons of the root are obtained as before by exchanging nontree edges with tree edges. The exchanges are made according to an order that ensures that the tree resulting from the exchange is the minimum spanning tree of the updated graph at the corresponding son. To ensure this, the nontree edges are considered for replacement in increasing order of weight. This is repeated at descendant nodes of the computation tree. The entire computation tree is generated in a branch and bound fashion. To ensure efficiency, we characterize each spanning tree generated by the exchange pair that generates it in the computation tree. The final algorithm is as follows: The generation algorithm first generates the tree at the root and the sons at the root are input to a queue indexed by the exchange pair. The actual sorted order is generated by selecting the minimum tree from among all queues. This is done by maintaining a priority queue containing the first element of each queue.

The algorithm that we describe is similar to *Gen*, but instead of traversing the nodes of the computation tree in a depth-first fashion, a branch and bound strategy is used where the node corresponding to the spanning tree that is to be output next is expanded.

ALGO Genwt
    Find min spanning tree, MST;
    Repeat
        Generate sons of node corresponding to MST
        by considering nontree edges in increasing weight order;
        Put each spanning tree generated into the
        queue indexed by the exchange using which it was
        obtained from its parent;
        Pick minimum weighted spanning trees, MST, from
        priority queue;
    Until all queues are empty.
END Genwt

3.1. Correctness and complexity.

LEMMA 3.1. *For node A in the computation tree, S_A is the Minimum Spanning Tree of G_A.*

Proof. The proof is by induction on the level of the tree. At the root, the claim is true by construction. Assume that the claim is true for a node A. The sons of the node are generated by considering nontree edges in increasing order of weight. Let the ordered set of nontree edges at A be $f_1, f_2, \ldots, f_m$. Let nontree edge f_i be used to generate sons $B_1, \ldots, B_k$ of A by exchanging with tree edges $e_1, \ldots, e_k$. Then edges $f_1, f_2, \ldots, f_{i-1}$ are absent from each of $G_{B_1}, G_{B_2}, \ldots, G_{B_k}$. Thus f_i is the smallest edge that is present in each of these graphs, but not in S_A. For each j from 1 to k, since e_j is absent from G_{B_j}, it follows that $S(G_{B_j})$ is the MST of G_{B_j}. □

LEMMA 3.2. *The number of exchanges is at most* $(V-1)(E-V+1)$.

Proof. The first entry in the exchange pair has $V-1$ values since the only edges allowed are those in the spanning tree associated with the root. This is true because the second entry in the exchange associated with any node is also in the IN set of that node and its descendants. The second entry in the exchange pair cannot be an edge in the spanning tree associated with the root, because first entry in the exchange associated with any node is in the OUT set of that node. The result follows. □

The next lemma follows from the branch and bound generation of the computation tree.

LEMMA 3.3. *The spanning trees enter the queue for each of the exchanges in sorted order.*

Thus we have the following theorem.

THEOREM 3.4. *Algorithm Genwt correctly sorts the nodes of the computation tree in* $O(N \log V + VE)$ *time.*

Proof. The correctness follows from Lemmas 3.1, 3.2 and 3.3. The time may be divided into generation of nodes in the computation tree and processing of the queues. Generation of the nodes in the computation tree requires $O(N+V+E)$ time. Processing the queues requires $O(N \log VE)$ steps since there are at most $O(VE)$ queues. An initial sorting of nontree edges may require $O(E \log E)$ steps resulting in the required time bound. □

THEOREM 3.5. *Genwt requires* $O(N+VE)$ *space.*

Proof. The proof follows from the space requirements of the computation tree and the queues. □

Output complexity. The output of the algorithm can be the computation tree itself in which the nodes are numbered according to the order in which they are sorted. Pointers to these nodes are maintained from a list whose indices give the index of the spanning tree. The space complexity is $O(N)$ words, where each word has $O(\log N)$ bits. The trees can also be explicitly listed out in $O(NV)$ time. Note that the output time thus exceeds the time for sorting.

4. Conclusions.

This paper presents a methodology for enumerating subgraphs of a given graph and illustrates this with the spanning tree problem. A companion paper [KKR94] describes enumeration of directed graphs. Efficient enumeration of cycles may also be possible using a similar scheme.

5. Acknowledgments.

We thank P. C. P. Bhatt, N. C. Kalra, S. N. Maheshwari, and S. Arun-Kumar for comments and pointers to references.

REFERENCES

[Ch68] J. P. CHAR, *Generation of trees, 2 trees and storage of master forests*, IEEE Trans. Circuit Theory, CT-15 (1968), pp. 128–138.

[Ga77] H. N. GABOW, *Two algorithms for generating weighted spanning trees in order*, SIAM J. Comput., 6 (1977), pp. 139–150.

[GM78] H. N. GABOW AND E. W. MYERS , *Finding all spanning trees of directed and undirected graphs*, SIAM J. Comput., 7 (1978).

[Ja89] A. JAMES, *A study of algorithms to enumerate all stable matchings and spanning trees*, M. Tech. thesis, Department of Mathematics, 1989, Indian Institute of Technology, Delhi, India.

[Jay81] R. JAYAKUMAR, *Analysis and study of a spanning tree enumeration algorithm*, Combinatorics and graph theory, Lecture Notes in Math., 885 Springer-Verlag, New York, 1981, pp. 284–289.

[JTS84] R. JAYARAMAN, M. THULASIRAMAN, AND M. N. S. SWAMY, *Complexity of computation of a spanning tree enumeration algorithm*, IEEE Trans. Circuits Systems, CAS-31 (1984), pp. 853–860.

[KJ89] N. C. KALRA AND S. S. JAMUAR, *Microprocessor based Char's tree enumeration algorithm*, JIETE, 35 (1989).

[KKR94] S. KAPOOR, V. KUMAR, AND H. RAMESH, *Faster enumeration of all spanning trees of a directed graph*, 1994, manuscript.

[KR92] S. KAPOOR AND H. RAMESH, *An algorithm for enumerating all spanning trees of a directed graph*, SIAM J. Comput., submitted.

[Ma72] W. MAYEDA, *Graph Theory*, John Wiley, New York, 1972, pp. 252–364.

[Mi65] G. J. MINTY, *A simple algorithm for listing all trees of a graph*, IEEE Trans. Circuit Theory, CT-12 (1965), pp. 120–125.

[Ra90] H. RAMESH, *An algorithm for enumerating all spanning trees of an undirected weighted graph in increasing order of weight*, manuscript.

[TR75] R. E. TARJAN AND R. C. READ, *Bounds on backtrack algorithms for listing cycles, paths and spanning trees*, Networks, 5 (1975), pp. 237–252.

SIAM J. COMPUT.
Vol. 24, No. 2, pp. 266–278, April 1995

006

PERMUTING IN PLACE*

FAITH E. FICH†, J. IAN MUNRO‡, AND PATRICIO V. POBLETE§

Abstract. This paper addresses the fundamental problem of permuting the elements of an array of n elements according to some given permutation. It aims to perform the permutation quickly by using only a polylogarithmic number of bits of extra storage. The main result is an algorithm whose worst case running time is $O(n \log n)$ and uses $O(\log n)$ additional $\log n$-bit words of memory. A simpler method is presented for the case in which both the permutation and its inverse can be computed at (amortised) unit cost. This algorithm requires $O(n \log n)$ time and $O(1)$ words in the worst case. These results are extended to the situation in which a power of the permutation must be applied. A linear time, $O(1)$ word method is presented for the special case in which the data values are all distinct and are either initially in sorted order or will be when permuted.

Key words. permutation, reordering, space, in place

AMS subject classifications. 68P05, 68P10

1. Introduction.

Given an array $A[1..n]$ and a permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$, we wish to rearrange the elements of the array in place according to the permutation. More precisely, the rearrangement consists of performing the equivalent of the n *simultaneous* assignments

$$A[\pi(i)] \leftarrow A[i] \text{ for } i \in \{1, \ldots, n\}.$$

Thus if the array A originally contains the sequence of values $a_1, a_2, \ldots, a_n$, then, afterwards, it contains the sequence $a_{\pi^{-1}(1)}, a_{\pi^{-1}(2)}, \ldots, a_{\pi^{-1}(n)}$.

"In place" means that the algorithm performs the rearrangement by repeatedly interchanging pairs of elements. Hence, the set of values in the array and the number of times each occurs always remain the same. In particular, this definition precludes moving the elements into an auxiliary array and then putting each element, one at a time, into its correct location in the original array. It also means that the array cannot be padded with blanks (see, for example, [7]) to make it a more convenient size to work with.

The problem of rearranging data arises in a variety of situations. Some examples are transposing a rectangular matrix [12, Ex. 1.3.3–12], [7], rotating a bit map image, and exchanging two sections of an array [2, p. 134].

In sorting a list with very large records, it might be more efficient to manipulate pointers to the records rather than the records themselves and, afterwards, put each record into its appropriate location [13, p. 74]. Another way to sort [13, p. 76] is to first compare every pair of keys, then, for each record, count the number of other records that should precede it and rearrange the records accordingly.

In certain circumstances, searching is *not* most efficient when the elements of the array are in sorted order. If a large number of searches are going to be performed, it could be advantageous to rearrange a sorted array first. For example, one can simplify the index computation in a binary search by organizing the elements into a data structure that implicitly represents a complete binary search tree (via a breadth first, left to right, enumeration of its nodes) [1, Ex. 12–6, pp. 136, 183–184], [13, Ex. 6.2.1–24, pp. 422, 670], [12, p. 401]. Like a

*Received by the editors October 13, 1992; accepted for publication June 10, 1993. This research was supported in part by the Natural Science and Engineering Research Council of Canada under grant numbers A-8237 and A-9176, the Information Technology Research Centre of Ontario, and the Chilean FONDECYT under grant numbers 90- 1263 and 91-1252.

†Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A4.

‡Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1.

§Departamento de Ciencias de la Computación, Universidad de Chile, Casilla 2777, Santiago, Chile.

heap, an element in location $i < n/2$ has its left and right children in locations $2i$ and $2i + 1$, respectively. When the data is stored on a disk with each block capable of holding b elements or in a virtual memory in which each page can contain b elements [13, pp. 472–473], the number of reads can be minimized by arranging the elements using a similar implicit representation of a complete $(b + 1)$-ary search tree [12, p. 401]. These three different arrangements are illustrated in Fig. 1. In the third case, $b = 3$.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

8	4	12	2	6	10	14	1	3	5	7	9	11	13	15

4	8	12	1	2	3	5	6	7	9	10	11	13	14	15

FIG. 1. *Three arrangements of a list of length* 15.

We are interested in the amount of time and additional space needed to rearrange the elements of the array according to the input permutation. Previously known algorithms for this problem used either quadratic time or a linear amount of additional space in the worst case. In this paper, we present a concise algorithm that takes time $O(n \log n)$ and uses only $O(\log^2 n)$ bits of additional storage. We have a simpler method for the case in which both the permutation and its inverse are given; it takes the same amount of time and uses only $O(\log n)$ bits. Furthermore, we show how to use the fact that a permutation is known to rearrange the array either to or from sorted order to obtain an algorithm that takes time $O(n)$ and uses only $O(\log n)$ bits of additional storage.

Throughout the paper, we assume that the permutation π is given by means of an oracle. This models the situation where the value of $\pi(i)$ is computed from i (e.g., transposing a rectangular array) or the permutation cannot be changed (e.g., because it is being used by other processes). If the permutation is given by an array and its entries can be used to record information as the algorithm proceeds (perhaps destroying the permutation in the process), data rearrangement can be done efficiently by using $O(n)$ time and $O(\log n)$ additional bits of storage [13, Ex. 5.2–10, pp. 80, 595]. This is also the case when the permutation is given as a product of disjoint cycles.

2. Cycle leader algorithms. The cycle structure of the permutation can be exploited to obtain efficient algorithms for rearranging data. Permutations are composed of one or more disjoint cycles, as illustrated in Fig. 2. The arrows follow the direction $i \rightarrow \pi(i)$, indicating the direction in which the data should flow.

i	1	2	3	4	5	6	7	8	9	10
$\pi(i)$	3	9	4	2	1	6	8	7	5	10

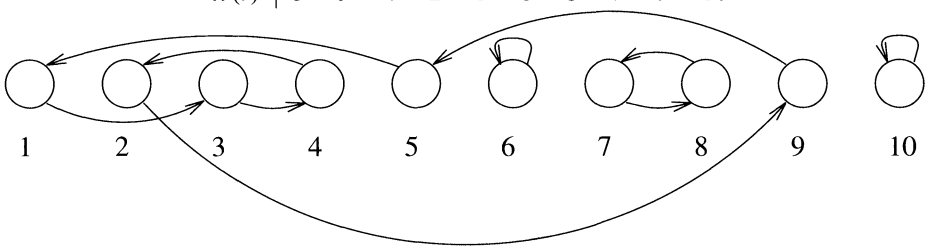


FIG. 2. *A graphical representation of a permutation.*

A basic operation is *ROTATE* (see Fig. 3) which, starting from some designated location in a cycle, moves the values from one location of the cycle to the next. We call this designated location the *cycle leader*. For example, the cycle leader could be the smallest location in the cycle. Once the cycle leader has been identified, *ROTATE* can be performed by proceeding through the cycle, starting with its cycle leader and exchanging the value located at the cycle leader with the values in the successive locations of the cycle. Each of these other values is involved in exactly one exchange (that takes it to its correct final location). Thus the time taken by such an algorithm is proportional to n, the length of the array, plus the amount of time it takes to find all the cycle leaders.

```
procedure ROTATE(leader)
i ← π(leader)
while i ≠ leader do
        interchange the values in A[leader] and A[i]
        i ← π(i)
```

FIG. 3. *Rotating the values in a cycle.*

If the permutation π is given as a product of disjoint cycles, identifying a leader for each cycle is very straightforward: merely take the first element in each cycle. The problem is more interesting when π is given as a mapping from the elements $1, \ldots, n$. One way to find all the cycle leaders in this case is to consider the locations $1, \ldots, n$ one at at time and, for each, determine whether it is a cycle leader.

If an additional, initially empty, bit vector of length n is available, it is easy to determine whether a location is the smallest element in its cycle in constant time. Specifically, when the value in an array location is moved, the bit corresponding to that location is set to 1. Since the locations are considered from smallest to largest, a location under consideration will have its corresponding bit equal to 0 exactly when it is a cycle leader [12, Ex. 1.3.3–12(b), pp. 180, 517–518]. Similarly, if π is given as an array whose elements can be modified, the same effect can be achieved by setting $\pi(i)$ to i when the value in array location i is moved.

Determining whether or not a location is the smallest element in its cycle can also be accomplished by using only a constant number of pointers into the array (each $\log_2 n$ bits long). Specifically, starting at the given location, proceed along the cycle until either the entire cycle has been traversed or a smaller location is encountered. In the first case, the given location is a cycle leader; otherwise it is not. The total amount of time to consider all n locations is $O(n^2)$. The worst case is achieved when $\pi = (1\ 2\ 3 \ldots n)$. For random permutations, an average of $O(n \log n)$ steps are performed [11], [13, p. 595].

These two ideas can be combined into the algorithm illustrated in Fig. 4.

THEOREM 2.1. *In the worst case, permuting an array of length n, given the permutation, can be done in* $O(n^2/b)$ *time and* $b + O(\log n)$ *bits of auxiliary space (consisting of a bit vector of length b plus a constant number of pointers) for* $b \leq n$.

The array A is conceptually divided into $\lceil n/b \rceil$ regions. Each region has size b, except for the last region, which might be smaller. The bit vector V is used to keep track of which locations in the region are encountered as the region is processed. If the location under consideration for being a cycle leader has a corresponding bit with value 0, its cycle is traversed until a smaller location is encountered. If no smaller location is encountered, then the location is a cycle leader and the cycle is rotated. Furthermore, if the location under consideration has a corresponding bit with value 1, then the location was previously encountered as part of a cycle containing some smaller location in the region and, hence, it is not a cycle leader. Theorem 2.1 follows from these observations.

```
for k ← 1 to ⌈n/b⌉ do
        s ← (k − 1)b
        if k ≤ ⌊n/b⌋ then l ← b
                     else l ← n − b⌊n/b⌋
        for i ← 1 to l do
            V[i] ← 0
        for i ← 1 to l do
            if V[i] = 0 then V[i] ← 1
                             j ← π(s + i)
                             while j > s + i do
                                 if j ≤ s + l then V[j − s] ← 1
                                 j ← π(j)
                             if j = s + i then ROTATE(s + i)
```

FIG. 4. *Rearranging an array using a bit vector V of length b.*

If both π and π^{-1} are available, then it is possible to get a more efficient algorithm. Determining whether or not a given location is the smallest element in its cycle can be done by starting at the location and proceeding alternatively forwards and backwards along the cycle until either the entire cycle has been traversed or a smaller location is encountered. (See Fig. 5.)

```
for i ∈ {1, ..., n} do
    j ← π(i)
    if j ≠ i then k ← π^{-1}(i)
                  while i < j and i < k do
                      if j = k then ROTATE(i)
                                    exit
                      j ← π(j)
                      if j = k then ROTATE(i)
                                    exit
                      k ← π^{-1}(k)
```

FIG. 5. *Rearranging an array using a permutation and its inverse.*

THEOREM 2.2. *In the worst case, permuting an array of length n, given the permutation and its inverse, can be done in $O(n \log n)$ time and $O(\log n)$ additional bits of storage.*

The analysis is similar to the bidirectional distributed algorithm for finding the smallest element in a ring of processors [8]. Specifically, the algorithm cannot determine whether i is a cycle leader after proceeding t steps forwards and t steps backwards along the cycle starting from i, only if the t elements following i and the t elements preceding i in its cycle are all larger than t. Furthermore, this will be the case for at most $1/t$ of the choices for i.

It is interesting to compare this algorithm to the $O(n^2)$ time $O(\log n)$ space algorithm mentioned above. Their expected behaviour on a random permutation is identical. However, the algorithm in Fig. 5 eliminates the bad cases.

It is not necessary that the cycle leader be the smallest (or largest) location in the cycle. One approach is to apply a hash function to the elements of the permutation and take as cycle leader the location in the cycle that hashes to the smallest value. Starting at a given location, the algorithm proceeds along the cycle until either the entire cycle has been traversed or another location that hashes to a smaller value is encountered. If the hash function is randomly chosen from among a set of 5-wise independent hash functions, then this results in a randomized algorithm using $O(\log n)$ space and expected time $O(n \log n)$ for every input permutation [9].

It was not clear to us whether or not it was possible to have a deterministic algorithm that used $n \log^{O(1)} n$ time and $\log^{O(1)} n$ space without having the inverse permutation available. However, using a rather different approach, we were able to devise such an algorithm.

THEOREM 2.3. *In the worst case, permuting an array of length n, given the permutation, can be done in $O(n \log n)$ time and $O(\log^2 n)$ additional bits of storage.*

First, consider the following characterization of the minimum locations in the cycles of the permutation π. Let $E_1 = \{1, \ldots, n\}$ and $\pi_1 = \pi$. For $r > 1$, we inductively define $E_r \subseteq E_{r-1}$ to be the set of *local minima* encountered following the permutation π_{r-1}, that is, $E_r = \{i \in E_{r-1} \mid \pi_{r-1}^{-1}(i) > i < \pi_{r-1}(i)\}$. We also define $\pi_r : E_r \to E_r$ to be the permutation that maps each element of E_r to the next element of E_r that is encountered following the permutation π_{r-1}. In other words, if $i \in E_r$, then $\pi_r(i) = \pi_{r-1}^m(i)$, where $m = \min\{m > 0 \mid \pi_{r-1}^m(i) \in E_r\}$, and $\pi_r(i) = \pi^M(i)$, where $M = \min\{M > 0 \mid \pi^M(i) \in E_r\}$. We call E_r the set of *order r elbows*.

For example, if $\pi = (1\ 5\ 3\ 6\ 10\ 4\ 2\ 9\ 8\ 7\ 11)$, as illustrated in Fig. 6, then

$E_1 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$, $\pi_1 = (1\ 5\ 3\ 6\ 10\ 4\ 2\ 9\ 8\ 7\ 11)$,
$E_2 = \{1, 2, 3, 7\}$, $\pi_2 = (1\ 3\ 2\ 7)$,
$E_3 = \{1, 2\}$, $\pi_3 = (1\ 2)$,
$E_4 = \{1\}$, and $\pi_4 = (1)$.

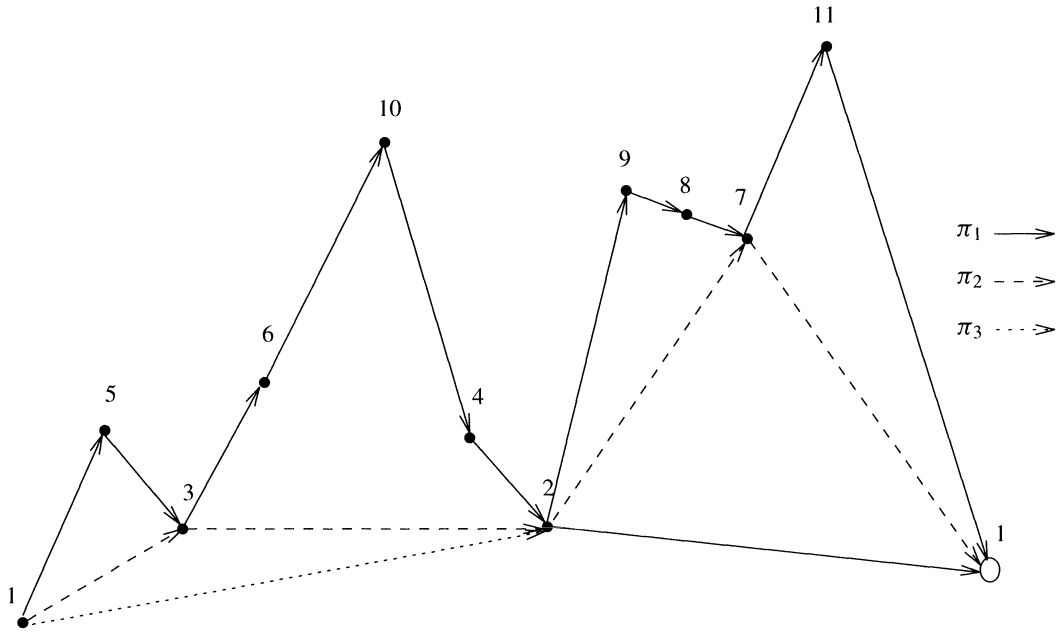


FIG. 6. *A cycle and its elbows.*

No more than half the elements in any cycle are local minima. Thus $|E_r| < |E_{r-1}|/2$. The minimum element of a cycle of π is always in E_r if the cycle contains more than one element of E_{r-1}. Hence the minimum element in a cycle is the unique elbow of maximum order in its cycle.

Computing the sets of elbows $E_1, E_2, \ldots$ and the corresponding permutations $\pi_1, \pi_2, \ldots$ can be done either in a top–down fashion, analogous to recursive descent parsing, or bottom–up, analogous to shift-reduce parsing. We explore both approaches in the two algorithms that follow.

The unidirectional nature of our oracle means that given $i \in E_r$, it is easy to compute $\pi_r(i)$ but difficult to compute $\pi_r^{-1}(i)$. Thus determining whether i is a local minimum of π_r (i.e., whether or not $i \in E_{r+1}$) is difficult starting from i, but easy starting from i's predecessor in π_r. For example, in Fig. 6, starting from 5 we can recognize that its successor, 3, is a local minimum in π_1. In turn, starting from 3 we can recognize that 2, its successor in π_2, is a local minimum in π_2.

For each cycle, the algorithm in Fig. 7 chooses the leader to be the unique element i in the cycle such that $\pi_{s-1} \ldots \pi_1(i) \in E_s$, where s is the maximum order of any element in the cycle. In other words, an element can be rejected as soon as it is seen not to be the element of maximum order. Dolev, Klawe, and Rodeh [4] and Peterson [14] independently discovered the usefulness of this choice of cycle leader for electing a leader in a unidirectional ring of processors using $O(n \log n)$ messages.

The procedure call $NEXT(r)$ is used to compute successive elements of π_r. It does this recursively, using successive elements of π_{r-1}.

For each $i \in \{1, \ldots, n\}$, the main program tests whether or not $\pi_1(i) \in E_2$ by comparing $\pi_1(i)$ with i and $\pi_1\pi_1(i)$; if so, it next tests whether or not $\pi_2\pi_1(i) \in E_3$ by comparing $\pi_2\pi_1(i)$ with $\pi_1(i)$ and $\pi_2\pi_2\pi_1(i)$, etc. One test is performed each iteration of the inner for loop. Eventually, a value of r is found such that $\pi_{r-1} \ldots \pi_1(i) \in E_r$ but $\pi_r\pi_{r-1} \ldots \pi_1(i) \notin E_{r+1}$. There are three possible ways this can occur. One is when $\pi_r\pi_{r-1} \ldots \pi_1(i) = \pi_{r-1} \ldots \pi_1(i)$. Then, since π_r is a permutation, this implies that the cycle of π_r that contains $\pi_{r-1} \ldots \pi_1(i)$ is trivial and thus $\pi_{r-1} \ldots \pi_1(i)$ is the minimum element in the cycle of π that contains it. In this case, i is a cycle leader. The other possible situations are when either $\pi_{r-1} \ldots \pi_1(i) < \pi_r\pi_{r-1} \ldots \pi_1(i)$ or $\pi_r\pi_{r-1} \ldots \pi_1(i) > \pi_r\pi_r\pi_{r-1} \ldots \pi_1(i)$. In both these cases, the algorithm detects that $\pi_r\pi_{r-1} \ldots \pi_1(i)$ is not the minimum element in its cycle and thus that i is not a cycle leader.

Using $O(\log^2 n)$ space, very little information about each permutation can be stored. We show that, essentially, only the most recently detected elbow of each order need be stored. The algorithm in Fig. 7 stores this information in the array $elbow$. For $r > 1$, $elbow[r]$ is used to store an element of E_r. Immediately prior to a call to $NEXT(r)$, the elements satisfy the condition

$$elbow[r] = elbow[r-1] \overset{\pi_{r-1}}{\rightarrow} \cdots \overset{\pi_1}{\rightarrow} elbow[0]$$

and immediately afterwards, they satisfy the condition

$$elbow[r] \overset{\pi_r}{\rightarrow} elbow[r-1] \overset{\pi_{r-1}}{\rightarrow} \cdots \overset{\pi_1}{\rightarrow} elbow[0].$$

The procedure $NEXT(r)$ computes $\pi_r(elbow[r])$. This leaves $elbow[r]$ unchanged and places the result in $elbow[r-1]$ (and, of course, updates the earlier elements in the array).

If $r > 1$, $NEXT(r)$ recursively computes successive elements along π_{r-1}, starting from $elbow[r]$, until a local minimum is detected. Since $elbow[r] \in E_r$, $elbow[r] < \pi_{r-1}(elbow[r])$. Initially, $elbow[r-1] = elbow[r]$ and $elbow[r-2] = \pi_{r-1}(elbow[r])$; thus $elbow[r-1] < elbow[r-2]$. The first while loop in $NEXT(r)$ advances $elbow[r-1]$ (and $elbow[r-2], \ldots, elbow[0]$, recursively) as long as π_{r-1} is strictly increasing. The cycle of π_{r-1} containing $elbow[r]$ is not trivial; thus, when the first while loop terminates, $elbow[r-1] > elbow[r-2] = \pi_{r-1}(elbow[r-1])$. The second while loop continues advancing $elbow[r-1]$ (and $elbow[r-2], \ldots, elbow[0]$, recursively) as long as π_{r-1} is strictly decreasing. At the end of the second while loop, $elbow[r-1] < elbow[r-2] = \pi_{r-1}(elbow[r-1])$. Hence $elbow[r-1]$ is the next local minimum of π_{r-1}. In other words, $elbow[r-1] = \pi_r(elbow[r])$.

Suppose $i' = \pi_r\pi_{r-1} \ldots \pi_1(i) \in E_r - E_{r+1}$. At the beginning of the rth iteration of the inner for loop of the main program, $elbow[r] = \pi_{r-1} \ldots \pi_1(i) = \pi_r^{-1}(i') \in E_r$. When the computation detects that $i' \notin E_{r+1}$, either $elbow[r] = \pi_r^{-1}(i')$ and $elbow[r-1] = i'$ or $elbow[r] = i'$ and $elbow[r-1] = \pi_r(i')$. Furthermore, $elbow[0] = \pi_1 \ldots \pi_{r-1}(elbow[r-1])$. It is easy to prove by induction on r that if $j \in E_r$ and M is the smallest positive integer such that $\pi_1 \ldots \pi_{r-1}(j) = \pi^M(j)$, then $\pi^L(j) \notin E_r$ for $1 \leq L \leq M$. Thus, starting from i' and proceeding along π, the element $\pi_1 \ldots \pi_{r-1}\pi_r(i')$ will be reached

```
procedure NEXT(r)
if r = 1 then elbow[0] ← π(elbow[1])
         else while elbow[r − 1] < elbow[r − 2] do
                    elbow[r − 1] ← elbow[r − 2]
                    NEXT(r − 1)
              while elbow[r − 1] > elbow[r − 2] do
                    elbow[r − 1] ← elbow[r − 2]
                    NEXT(r − 1)
return

for i ∈ {1, ..., n} do
      elbow[0] ← elbow[1] ← i
      for r ← 1, 2, ... do
            {loop invariant: elbow[r] = π_{r−1} ··· π_2 π_1(i) ∈ E_r}
            NEXT(r)
            if elbow[r] > elbow[r − 1]
            then elbow[r] ← elbow[r − 1]
                 NEXT(r)
                 if elbow[r] > elbow[r − 1] then exit
                 elbow[r + 1] ← elbow[r]
            else if elbow[r] = elbow[r − 1] then ROTATE(i)
                 exit
```

FIG. 7. *A recursive algorithm that rearranges an array using $O(n \log n)$ time and $O(\log^2 n)$ space.*

before $\pi_r^2(i')$. Similarly, starting from i' and proceeding backwards along π, the element $i = \pi_1^{-1} \ldots \pi_{r-1}^{-1} \pi_r^{-1}(i')$ will be reached before $\pi_r^{-2}(i')$. Therefore, testing whether or not i is a cycle leader involves proceeding along π in a subsegment of the region between $\pi_r^{-2}(i')$ and $\pi_r^2(i')$.

Considering all $i' \in E_r - E_{r+1}$, the permutation π is evaluated less than $4n$ times. (In fact, π is evaluated at most 4 times at each element.) Since E_r is empty for $r > \log_2 n$, the algorithm proceeds a total of $O(n \log n)$ steps along π.

Each call of $NEXT(1)$ proceeds one step along π. Each call of $NEXT(r)$, for $r > 1$, involves at least two recursive calls to $NEXT(r-1)$. The total amount of work performed by this call (excluding the work performed by the recursive calls) is proportional to the number of recursive calls it makes. For each i, every iteration of the inner for loop performs a constant amount of work (excluding the work performed by the subroutines it calls) and, except for possibly the last iteration, involves two calls to *NEXT*. Thus the total amount of work performed by the entire algorithm is proportional to the total number of steps the algorithm proceeds along π plus $O(n)$ steps to rotate all the cycles.

When π consists of a single cycle with the elements $\{0, \ldots, n-1\}$ ordered lexicographically with respect to the reverse of their $(\log n)$-bit binary representations (for example, $\pi = (0\ 4\ 2\ 6\ 1\ 5\ 3\ 7)$), the algorithm actually uses $\Omega(n \log n)$ steps. Hence the running time of the algorithm is in $\Theta(n \log n)$.

The algorithm uses $\Theta(\log n)$ variables, each capable of holding one element in $\{1, \ldots, n\}$. With care in implementation, only $O(\log n)$ bits are needed for representing the program stack. Thus a total of $\Theta(\log^2 n)$ bits of additional space are used by this algorithm.

An iterative, bottom–up version of the algorithm in Fig. 7 is given in Fig. 8. As $elbow[0]$ is advanced along the cycle of π containing i, $elbow[r]$, for $r > 1$, records the most recent element of E_r that has been detected. Each time a local minimum along π_{r-1} is detected, $elbow[r]$ is advanced to that location. The variable $state[r]$ encodes information about the portion of the cycle of π_r that is being examined.

The first time an element of E_r is detected, $state[r]$ is set to *GOT_ONE* and $elbow[r] = \pi_{r-1} \ldots \pi_1(i)$. When the next element of E_r (along π_r) is detected, the algorithm can determine

```
for i ∈ {1, ..., n} do
        elbow[1] ← i
        state[1] ← GOT_ONE
        repeat
                r ← 1
                elbow[0] ← π(elbow[1])
                while state[r] =DOWN and elbow[r] < elbow[r − 1] do
                        state[r] ←UP
                        elbow[r] ← elbow[r − 1]
                        r ← r + 1
                case state[r]

                GOT_ONE: if elbow[r] > elbow[r − 1]
                        then    state[r] ← GOT_TWO
                                elbow[r] ← elbow[r − 1]
                        else    if elbow[r] = elbow[r − 1] then ROTATE(i)
                                exit

                GOT_TWO: if elbow[r] > elbow[r − 1] then exit
                        state[r + 1] ← GOT_ONE
                        elbow[r + 1] ← elbow[r]
                        state[r] ← UP
                        elbow[r] ← elbow[r − 1]

                UP:     if elbow[r] > elbow[r − 1] then state[r] ← DOWN
                        elbow[r] ← elbow[r − 1]

                DOWN:   elbow[r] ← elbow[r − 1]
```

FIG. 8. *An iterative algorithm that rearranges an array using $O(n \log n)$ time and $O(\log^2 n)$ space.*

whether i's cycle contains only one element in E_r. If so, i is the leader of its cycle. Otherwise the algorithm sets $state[r]$ to GOT_TWO and tries to determine whether or not $\pi_r \pi_{r-1} \ldots \pi_1(i)$ is a local minimum of π_r (and hence an element of E_{r+1}).

If $state[r] = UP$, then $elbow[r]$ is known to be larger than $\pi_r^{-1}(elbow[r])$, which is its predecessor along π_r. Similarly, if $state[r] = DOWN$, then $elbow[r]$ is known to be smaller than $\pi_r^{-1}(elbow[r])$, which is its predecessor along π_r. This information is sufficient to detect successive local minima along π_r. Figure 9 illustrates the states that occur as the algorithm proceeds along a cycle.

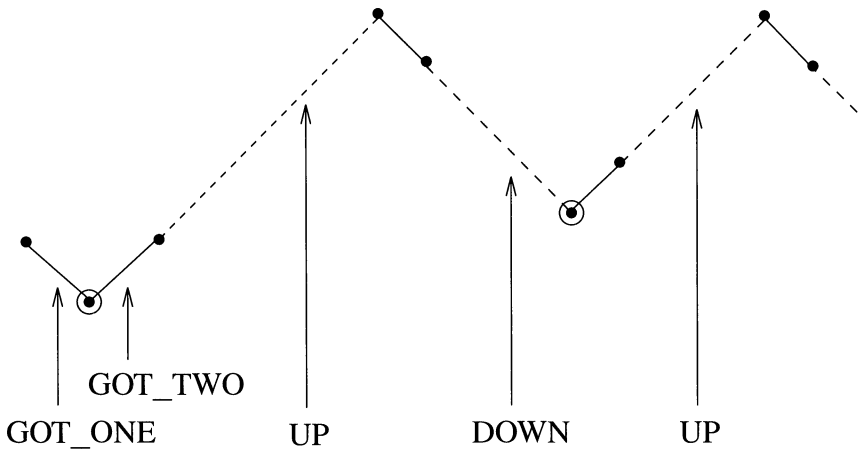


FIG. 9. *The states occurring in the iterative algorithm.*

```
procedure ROTATE_BACKWARDS(leader)
j ← leader
i ← π(leader)
while i ≠ leader do
        interchange the values in A[j] and A[i]
        j ← i
        i ← π(i)
```

FIG. 10. *Rotating the values backwards in a cycle.*

```
procedure EXCHANGE(i, j, m)
repeat m times
        interchange the values in A[j] and A[i]
        i ← π(i)
        j ← π(j)

procedure ROTATE(q, leader, cyclelength)
irun ← q mod cyclelength
jrun ← cyclelength − irun
i ← leader
j ← leader
repeat irun times
        j ← π[j]
while irun ≠ jrun do
        if irun < jrun
        then temp ← i
                EXCHANGE(i, j, irun)
                i ← temp
                jrun ← jrun − irun
        else    temp ← i
                EXCHANGE(i, j, jrun)
                j ← temp
                irun ← irun − jrun
EXCHANGE(i, j, irun)
```

FIG. 11. *Rotating the values q steps in a cycle.*

A small but natural twist on our problem is that we are given π, but required to move the elements according to π^{-1}. The cycle leader technique of Theorem 2.3 is still applicable. It is only necessary to replace *ROTATE* by the procedure *ROTATE_BACKWARDS* given in Fig. 10.

Clearly this step requires time linear in the size of the cycle, and $O(\log n)$ bits of memory. The problem is easily generalized to being given π (and perhaps π^{-1}) and asked to apply π^q, where q may depend on the (length of the) cycle. Either one of the cycle leader methods is applicable. We need only define a modified procedure *ROTATE*$(q, leader, cyclelength)$. This is essentially the same problem as transposing an array. The following is a translation into our terms of the method discussed in [2, p. 134]. Suppose that the m elements along π starting at location i are distinct from the m elements along π starting at location j. Then the procedure *EXCHANGE*(i, j, m) exchanges these two sequences of elements. Both i and j are advanced m locations along the cycle as a result of an execution of the procedure *EXCHANGE* in Fig. 11.

We see that, including calls to π, the *ROTATE* algorithm requires time proportional to the length of the cycle in question; hence we can strengthen Theorems 2.2 and 2.3.

COROLLARY 2.4. *In the worst case, permuting an array of length n according to the permutation π^q, given π and its inverse, can be done in $O(n \log n)$ time and $O(\log n)$ additional bits of storage.*

COROLLARY 2.5. *In the worst case, permuting an array of length n according to the permutation π^q, given the permutation π, can be done in $O(n \log n)$ time and $O(\log^2 n)$ additional bits of storage.*

3. Permuting data in and out of order. In many situations, it is known that the array elements satisfy a fixed total order, i.e., when the data values are taken in a particular order, they form a sorted sequence. Suppose the array A has been rearranged according to the permutation π and let σ be a permutation such that

$$A[\sigma^{-1}(1)] \leq A[\sigma^{-1}(2)] \leq \cdots \leq A[\sigma^{-1}(n)].$$

Then element $A[i]$ has the $\sigma(i)$th smallest value, for $i = 1, \ldots, n$. In particular, if $\sigma(i) > \sigma(\pi(i))$ then $A[i] > A[\pi(i)]$. For example, if the permutation π rearranges the array A into sorted order, then σ is the identity permutation and if A was sorted before the rearrangement, then $\sigma = \pi^{-1}$.

Consider the rearrangement algorithm in Fig. 12. In particular, if the permutation π rearranges the array A into sorted order, then the algorithm is simply looking for places where the permutation wants to move an element to a lower numbered location and the element is smaller than the element that is currently there.

```
for i ∈ {1, ..., n} do
        if (σ(i) > σ(π(i))) and (A[i] < A[π(i)])
                then ROTATE(i)
```

FIG. 12. *Rearranging an array of distinct elements that satisfy a fixed total order.*

Notice that after the cycle containing location i has been rotated, $\sigma(i) > \sigma(\pi(i))$ implies $A[i] \geq A[\pi(i)]$. Thus each nontrivial cycle is rotated at most once.

Now suppose that no two consecutive elements along any nontrivial cycle have the same value, i.e., $i \neq \pi(i)$ implies $A[i] \neq A[\pi(i)]$ for all $i \in \{1, \ldots, n\}$. This condition is certainly true if the elements of the array A are distinct. We claim that before a given nontrivial cycle of the permutation π has been rotated, there is a location i in that cycle such that $\sigma(i) > \sigma(\pi(i))$ and $A[i] < A[\pi(i)]$. This implies that the cycle will be rotated at least once. Let i be any location in the cycle that satisfies $\sigma(i) > \sigma(\pi(i)) < \sigma(\pi^2(i))$. (This will be the case, for example, when $\pi(i)$ is the location containing the smallest value in the cycle.) The cycle leader will be the first such location encountered. After the cycle has been rotated, $A[\pi(i)] \leq A[\pi^2(i)]$, since $\sigma(\pi(i)) < \sigma(\pi^2(i))$. Also, the values in locations $\pi(i)$ and $\pi^2(i)$ were initially in locations i and $\pi(i)$, respectively. Hence, before the rotation, $A[i] \leq A[\pi(i)]$. However, $A[i] \neq A[\pi(i)]$, so $A[i] < A[\pi(i)]$, as required.

It is not necessary to rotate a cycle if all the elements it contains have the same value. However, there are other situations where a nontrivial cycle contains consecutive elements with the same value and the algorithm in Fig. 12 does not rotate the cycle. The example in Fig. 13 is constructed using two distinct values, α and β, where $\alpha < \beta$. It assumes the array is to be rearranged into sorted order. Except for the two middle elements, the array is already sorted, but since these two elements can be arbitrarily far apart in the cycle, no local test (such as the one in the algorithm in Fig. 12) can succeed in detecting them.

Notice that a cycle containing runs of equal-valued elements has the same effect as the cycle taking the beginning of each run to the beginning of the next run. Since no consecutive elements of this latter cycle have the same value, the technique used by the algorithm in Fig. 12 can be applied to identify a cycle leader.

If the functions π and $\sigma\pi$ can be evaluated in constant time (which is the case when π rearranges the array either into or out of sorted order), then the algorithm in Fig. 14 runs in

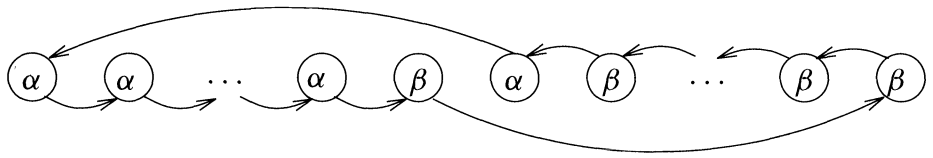


FIG. 13. *An example with runs of equal-valued elements.*

linear time. This is because the algorithm goes through the while loop at most once for each run of equal-valued elements in a cycle. In fact, the linear time bound holds also under a weaker assumption.

for $i \in \{1, \ldots, n\}$ **do**
  if $A[i] \neq A[\pi(i)]$ **then**
    $j \leftarrow \pi(i)$
    while $A[j] = A[\pi(j)]$ **do** $j \leftarrow \pi(j)$
    if $(\sigma(\pi(i)) > \sigma(\pi(j)))$ **and** $(A[\pi(i)] < A[\pi(j)])$ **then** *ROTATE*(i)

FIG. 14. *Rearranging any array of elements that satisfy a fixed total order.*

THEOREM 3.1. *Suppose that after the array A is permuted according to the permutation* π, *it satisfies* $A[\sigma^{-1}(1)] \leq A[\sigma^{-1}(2)] \leq \cdots \leq A[\sigma^{-1}(n)]$. *If all the elements* $\pi(1), \ldots, \pi(n)$, $\sigma\pi(1), \ldots, \sigma\pi(n)$ *can be computed in time* $O(n)$, *then A can be permuted according to* π *by using* $O(n)$ *time and* $O(\log n)$ *additional bits of storage, in the worst case.*

The proof follows from the observation that, for each element i, $\pi(i)$ and $\sigma\pi(i)$ are computed only a constant number of times during the course of the algorithm in Fig. 14. The running time of this algorithm is clearly dominated by the time spent performing these function evaluations.

The application that sparked our interest in the problem of rearranging data was building a data structure implicitly representing a complete binary search tree, given a sorted array of appropriate length n. Here the median is the root and is placed in location 1. Its children, the first and third quartiles, are in locations 2 and 3, respectively. In general, the left and right children of the element in location j are in locations $2j$ and $2j + 1$ (like a heap). Bentley [1] anticipates the result shown here in crediting Mahaney and Munro with a linear algorithm for this special case. In this case, the permutation π satisfies the property that if $x10^j$ is the $\lceil \log_2 n \rceil$-bit binary representation of i, then $0^j 1x$ is the $\lceil \log_2 n \rceil$-bit binary representation of $\pi(i)$.

When $n = 2^h - 1 + r$, where $1 \leq r < 2^h$, the sorted array can be rearranged to implicitly represent a binary search tree which is complete except for its last level and such that the nodes in the last level are as far left as possible. One method is to apply two permutations to the array. The first permutation π' moves the r values $1, 3, 5, \ldots, 2r - 1$ that belong in the last level of the tree to their correct locations at the end of the array, while keeping the other values in sorted order. Specifically,

$$\pi'(i) = \begin{cases} 2h + (i-1)/2 & \text{if } i \text{ is odd and } i < 2r, \\ i/2 & \text{if } i \text{ is even and } i < 2r, \\ i - r & \text{if } i > 2r. \end{cases}$$

The second permutation then rearranges the first $2^h - 1$ elements of the array into the implicit representation of a complete binary search tree, as above. Another method is to combine these two permutations into one, defining π as follows.

if i is odd **and** $i < 2r$
then $\pi(i) = 2^h + (i-1)/2$
else $\pi(i)$ is the number represented by $0^j 1x$,
where $x10^j$ is the h-bit binary representation of $\begin{cases} i/2, & \text{if } i \text{ is even and } i < 2r, \\ i - r, & \text{if } i \geq 2r. \end{cases}$

For most instruction sets, computing π once could take as much as $O(\log n)$ steps in the worst case. However, using only left and right shifts of distance one and single bit assignments and tests, $O(n)$ operations suffice to evaluate $\pi(i)$ for all $i \in \{1, \ldots, n\}$. As mentioned above, this is sufficient to obtain a linear time algorithm.

4. Conclusions. Since most permutations fix very few of their elements, any algorithm must take $\Omega(n)$ time on average and, hence, in the worst case. Also, $\Omega(\log n)$ additional bits of storage are needed to provide pointers to array elements that are being interchanged.

The open question that remains is whether there are better algorithms for rearranging an array given an arbitrary permutation π (and also possibly π^{-1}). Specifically, are there algorithms that use a small amount of additional space (e.g., $O(\log n)$ or $\log^{O(1)} n$) and only linear time? If not, are there deterministic algorithms that, given only π, run in $O(n \log n)$ time but use only $O(\log n)$ bits of additional storage?

When $b = n/\log\log n$, the algorithm in Fig. 4 uses $O(n \log\log n)$ time and $O(n/\log\log n)$ bits of additional storage. However, it is not even known whether there is an algorithm that uses $o(n \log n)$ time and $O(n^{1-\epsilon})$ space for some constant $\epsilon > 0$.

All of the algorithms presented in this paper are cycle leader algorithms. They proceed by identifying exactly one element in each cycle and rotating the cycle starting from that element. Performing all the rotations requires only linear time and two pointers. This leaves the problem of finding a leader for each cycle, or equivalently, finding the minimum element in each cycle. Cook and McKenzie [3] have shown that these problems and, more generally, computing the disjoint cycle representation of a permutation are NC^1-complete for deterministic logspace.

A previous version of the algorithm in Fig. 7 was interesting from the point of view that it did not operate by determining cycle leaders. Instead, it performed sweeps up and down the array, at each point deciding whether or not to interchange that element with another element and, if so, which one. It would be interesting to know if any algorithm for permuting an array can be transformed into a cycle leader algorithm. This would allow us to restrict attention to cycle leader algorithms when searching for new algorithms or trying to prove better tradeoffs.

The algorithms in Figs. 5, 7, and 8 and the $O(n^2)$ time, $O(\log n)$ space algorithm discussed before Fig. 4 can all be viewed as instances of a restricted type of cycle leader algorithm. Specifically, they separately test each element $i \in \{1, \ldots, n\}$ to see if it is a cycle leader by comparing elements forwards and/or backwards along the cycle. The tests depend only on the permutation π and the values of i. They do not depend on the order in which the elements are tested, as is the case for the algorithm in Fig. 4 (when $b > 1$) and the algorithm in Fig. 14, nor on the data values stored in the array, as is the case for the algorithm in Fig. 14.

Such algorithms are interesting because they immediately lead to distributed algorithms, using only comparisons of ID's, for electing a leader in a bidirectional ring of synchronous processors. Processor i tests whether element i is the leader of its cycle once it has learned the ID's of a sufficient number of its successors and predecessors. It requests these ID's by sending messages forwards and backwards around the ring for distances which are successive powers of 2. (See [8].) The total number of messages sent is proportional to the time taken by the cycle leader algorithm. Frederickson and Lynch [5] hava shown that any such distributed leader election algorithm sends $\Omega(n \log n)$ messages in the worst case. Thus these restricted types of cycle leader algorithms must use at least $\Omega(n \log n)$ time in the worst case. On the other hand, it is not clear how algorithms for electing a leader in a ring of processors can be

used to obtain cycle leader algorithms. The major problems seem to be how to deal with the timing of messages and how to represent the states of all the processors using less than a linear amount of space.

REFERENCES

[1] J. L. BENTLEY, *Programming Pearls*, Addison-Wesley, Reading, Mass., 1986.
[2] G. BRASSARD AND P. BRATLEY, *Algorithmics*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.
[3] S. COOK AND P. MCKENZIE, *Problems complete for deterministic logarithmic space*, J. Algorithms, 8 (1987), pp. 385–394.
[4] D. DOLEV, M. KLAWE, AND M. RODEH, *$O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle*, J. Algorithms, 3 (1982), pp. 245–260.
[5] G. N. FREDERICKSON AND N. A. LYNCH, *The impact of synchronous communication on the problem of electing a leader in a ring*, in Proc. 16th ACM Symposium on Theory of Computing, ACM Press, New York, 1984, pp. 493–505.
[6] R. G. GALLAGER, P. A. HUMBLET, AND P. M. SPIRA, *A distributed algorithm for minimum-weight spanning trees*, ACM Trans. Prog. Lang. Sys., 5 (1983), pp. 66–77.
[7] G. C. GOLDBOGEN, *PRIM: A fast matrix transpose method*, IEEE Trans. Soft. Eng., SE-7 (1981), pp. 255–257.
[8] D. S. HIRSCHBERG AND J. B. SINCLAIR, *Decentralized extrema-finding in circular configurations of processes*, Communications of the Association for Computing Machinery, 23 (1980), pp. 627–628.
[9] R. IMPAGLIAZZO, private communication.
[10] W. W. KIRCHHERR, *Transposition of an $l \times l$ matrix requires $\Omega(\log l)$ reversals on conservative turing machines*, Inform. Process. Lett., 28 (1988), pp. 55–59.
[11] D. E. KNUTH, *Mathematical Analysis of Algorithms*, Proc. IFIP Congress, 1 (1971), pp. 19–27.
[12] ———, *The Art of Computer Programming, Vol.* 1: *Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1973.
[13] ———, *The Art of Computer Programming, Vol.* 3: *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.
[14] G. L. PETERSON, *An $O(n \log n)$ Unidirectional algorithm for the circular extrema problem*, ACM Trans. Prog. Lang. Sys., 4 (1982), pp. 758–762.

SIAM J. COMPUT.
Vol. 24, No. 2, pp. 279–295, April 1995

007

THE COMPLEXITY AND DISTRIBUTION OF HARD PROBLEMS*

DAVID W. JUEDES† AND JACK H. LUTZ‡

Abstract. Measure-theoretic aspects of the $\leq_m^P$-reducibility structure of the exponential time complexity classes E=DTIME(2^{linear}) and E_2 = DTIME($2^{polynomial}$) are investigated. Particular attention is given to the *complexity* (measured by the size of complexity cores) and *distribution* (abundance in the sense of measure) of languages that are $\leq_m^P$-hard for E and other complexity classes.

Tight upper and lower bounds on the size of complexity cores of hard languages are derived. The upper bound says that the $\leq_m^P$-hard languages for E are *unusually simple*, in the sense that they have smaller complexity cores than most languages in E. It follows that the $\leq_m^P$-complete languages for E form a measure 0 subset of E (and similarly in E_2).

This latter fact is seen to be a special case of a more general theorem, namely, that *every* $\leq_m^P$-degree (e.g., the degree of all $\leq_m^P$-complete languages for NP) has measure 0 in E and in E_2.

Key words. computational complexity, complexity classes, complete problems, complexity cores, polynomial reducibilities, resource-bounded measure

AMS subject classification. 68Q15

1. Introduction.

A decision problem (i.e., language) $A \subseteq \{0,1\}^*$ is said to be *hard* for a complexity class $\mathcal{C}$ if every language in $\mathcal{C}$ is efficiently reducible to A. If A is also an element of $\mathcal{C}$, then A is *complete* for $\mathcal{C}$. The most common interpretation of "efficiently reducible" here is "polynomial time many-one reducible," abbreviated "$\leq_m^P$-reducible." (See §2 for notation and terminology used in this introduction.) For example, in most usages, "NP-complete" means "$\leq_m^P$-complete for NP," the completeness notion introduced by Karp [15] and Levin [16].

In this paper, we investigate the *complexity* (measured by size of complexity cores) and *distribution* (i.e., abundance in the sense of measure) of languages that are $\leq_m^P$-hard for E (equivalently, E_2) and other complexity classes, including NP. (By "measure" here, we mean *resource-bounded measure* as developed by Lutz [17] and described in §3 of this paper.) We give a tight lower bound and, perhaps surprisingly, a tight *upper* bound on the sizes of complexity cores of hard languages. More generally, we analyze measure-theoretic aspects of the $\leq_m^P$-reducibility structure of exponential time complexity classes. We prove that $\leq_m^P$-hard problems are rare, in the sense that they form a p-measure 0 set. We also prove that every $\leq_m^P$-degree has measure 0 in exponential time.

Complexity cores, first introduced by Lynch [24] have been studied extensively [8]–[12], [14], [27]–[29]. Intuitively, a complexity core of a language A is a fixed set K of inputs such that *every* machine whose decisions are consistent with A fails to decide efficiently on all but finitely many elements of K. The meaning of "efficiently" is a parameter of the definition that varies according to the context. (See §4 for a precise definition.)

Orponen and Schöning [28] have established two lower bounds on the sizes of complexity cores of hard languages. First, every $\leq_m^P$-hard language for E has a dense P-complexity core. Second, if P $\neq$ NP, then every $\leq_m^P$-hard language for NP has a nonsparse polynomial complexity core.

*Received by the editors October 8, 1992; accepted for publication (in revised form) September 30, 1993. This research was supported in part by National Science Foundation grants CCR-8809238 and CCR-9157382, with matching funds from Rockwell International and Microware Systems Corporation, and in part by the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS), where the second author was a visitor while part of this work was carried out.

†Department of Computer Science, Iowa State University, Ames, Iowa 50011. Present address, Department of Computer Science, Ohio University, Morton Hall 416, Athens, Ohio 45701.

‡Department of Computer Science, Iowa State University, Ames, Iowa 50011.

In §4 below, we extend the first of these results to languages that are weakly $\leq_m^P$-hard for E. (A language A is $\leq_m^P$-hard for E if every element of E is $\leq_m^P$-reducible to A. A language A is *weakly* $\leq_m^P$*-hard* for E if every element of some nonnegligible (i.e., nonmeasure 0) set of languages in E is reducible to A. Very recently, Lutz [19] has proven that "weakly $\leq_m^P$-hard" is more general than "$\leq_m^P$-hard.") Specifically, we prove that every language that is weakly $\leq_m^P$-hard for E or E_2 has a dense exponential complexity core. It follows that if NP does not have measure 0 in E or E_2, then every $\leq_m^P$-hard language for NP has a dense exponential complexity core. This conclusion is much stronger than Orponen and Schöning's conclusion that every such language has a nonsparse polynomial complexity core, although it is achieved at the cost of a stronger hypothesis. This hypothesis, originally proposed by Lutz, is discussed at some length in [18], [23], and [22].

In §5 we investigate the resource-bounded measure of the lower $\leq_m^P$-spans, the upper $\leq_m^P$-spans, and the $\leq_m^P$-degrees of languages in E and E_2. (The *lower* $\leq_m^P$*-span* of A is the set of all languages that are $\leq_m^P$-reducible to A. The *upper* $\leq_m^P$*-span* of A is the set of all languages to which A is $\leq_m^P$-reducible. The $\leq_m^P$*-degree* of A is the intersection of these two spans.) We prove the Small Span Theorem, which says that if A is in E or E_2, then at least one of the upper and lower spans must have resource-bounded measure 0. This implies that *every* $\leq_m^P$-degree (e.g., the degree of all $\leq_m^P$-complete languages for NP) has measure 0 in E and in E_2. It also implies that the $\leq_m^P$-hard languages for E form a set of p-measure 0. As noted in §7, a proof that this latter fact holds with $\leq_m^P$ replaced by $\leq_T^P$ would imply that $E \not\subseteq BPP$.

Languages that are $\leq_m^P$-hard for E are typically considered to be "at least as complex as" any element of E. Very early, Berman [6] established limits to this interpretation by proving that no $\leq_m^P$-complete language is P-immune, even though E contains P-immune languages. (In fact, Mayordomo [25] has recently shown that almost every language in E is P-bi-immune.) In §6 below we prove a very strong limitation on the complexity of $\leq_m^P$-hard languages for E. We prove that every $\leq_m^P$-hard language for E is decidable in $\leq 2^{4n}$ steps on a dense set of inputs which is also decidable in $\leq 2^{4n}$ steps. This implies that *every* DTIME(2^{4n})-complexity core of *every* $\leq_m^P$-hard language for E has a *dense complement*. Since almost every language in E has $\{0, 1\}^*$ as a DTIME(2^{4n})-complexity core (as proven in section 4), this says that $\leq_m^P$-hard languages for E are *unusually simple*, in that they have *unusually small* complexity cores. Intuitively, we interpret this to mean that the condition of being $\leq_m^P$-hard for E forces a language to have a high level of organization, thereby forcing it to be unusually simple in some respects.

2. Preliminaries. Here we present the notation and terminology that we use throughout the paper. To begin with, we write $\mathbf{N}$ for the set of natural numbers, $\mathbf{Z}$ for the set of integers, and $\mathbf{Z}^+$ for set of positive integers.

We deal primarily with *strings*, *languages*, *functions*, and *classes*. Strings are finite sequences of characters over the alphabet $\{0, 1\}$; we write $\{0, 1\}^*$ for the set of all strings. Languages are sets of strings. Functions usually map $\{0, 1\}^*$ into $\{0, 1\}^*$. A class is either a set of languages or a set of functions.

If $x \in \{0, 1\}^*$ is a string, we write $|x|$ for the *length* of x. If $A \subseteq \{0, 1\}^*$ is a language, then we write A^c, $A_{\leq n}$, and $A_{=n}$ for $\{0, 1\}^* - A$, $A \cap \{0, 1\}^{\leq n}$, and $A \cap \{0, 1\}^n$, respectively. The sequence of strings over $\{0, 1\}$, $s_0 = \lambda, s_1 = 0, s_2 = 1, s_3 = 00, \ldots$, is referred to as the *standard enumeration* of $\{0, 1\}^*$.

We use the string-pairing function $\langle x, y\rangle = bd(x)01y$, where $bd(x)$ is x with each bit doubled (e.g., $bd(1101) = 11110011$). For each $g : \{0, 1\}^* \to \{0, 1\}^*$ and $k \in \mathbf{N}$, we also define the function $g_k : \{0, 1\}^* \to \{0, 1\}^*$ by $g_k(x) = g(\langle 0^k, x\rangle)$ for all $x \in \{0, 1\}^*$.

We say that a property $\phi(n)$ of natural numbers holds *almost everywhere* (*a.e.*) if $\phi(n)$ is true for all but finitely many $n \in \mathbf{N}$. Similarly, $\phi(n)$ holds *infinitely often* (*i.o.*) if $\phi(n)$ is

true for infinitely many $n \in \mathbf{N}$. We write $[\![\phi]\!]$ for the Boolean value of a condition ϕ. That is, $[\![\phi]\!] = 1$ if ϕ is true, $[\![\phi]\!] = 0$ if ϕ is false.

If A is a finite set, we denote its cardinality by $|A|$. A language D is *dense* if there exists some constant $\epsilon > 0$ such that $|D_{\leq n}| > 2^{n^\epsilon}$ a.e. A language S is *sparse* if there exists a polynomial p such that $|S_{\leq n}| \leq p(n)$ a.e.. A language S is *cosparse* if S^c is sparse.

All *machines* here are deterministic Turing machines. The language accepted by a machine M is denoted by $L(M)$. The partial function computed by a machine M is denoted by $f_M : \{0,1\}^* \to \{0,1\}^*$. For a fixed machine M, the function $\text{time}_M(x)$ represents the number of steps that M uses on input x.

If $t(n)$ is a time bound, then we write

$$\text{DTIME}(t(n)) = \{L(M) \mid (\exists c)(\forall x)\text{time}_M(x) \leq c \cdot t(|x|) + c\}$$

for the set of languages decidable in $O(t(n))$ time. Similarly, we write

$$\text{DTIMEF}(t(n)) = \{f_M \mid (\exists c)(\forall x)\text{time}_M(x) \leq c \cdot t(|x|) + c\}$$

for the set of functions computable in $O(t(n))$-time. The classes of polynomial time decidable languages and polynomial time computable functions are then $\text{P} = \cup_{k=0}^{\infty}\text{DTIME}(n^k)$ and $\text{PF} = \cup_{k=0}^{\infty}\text{DTIMEF}(n^k)$, respectively. We are especially interested in classes of languages decidable in exponential time. We write

$$\text{E} = \bigcup_{c=1}^{\infty}\text{DTIME}(2^{cn})$$

and

$$\text{E}_2 = \bigcup_{c=1}^{\infty}\text{DTIME}(2^{n^c})$$

for the classes of languages decidable in 2^{linear} time and $2^{\text{polynomial}}$ time, respectively. Other complexity classes that we use here, such as NP, PH, PSPACE, etc., have completely standard definitions [2], [3].

If A and B are languages, then a *polynomial time, many-one reduction* (briefly, $\leq_{\text{m}}^{\text{P}}$-*reduction*) *of* A *to* B is a function $f \in \text{PF}$ such that $A = f^{-1}(B) = \{x \mid f(x) \in B\}$. A $\leq_{\text{m}}^{\text{P}}$-*reduction of* A is a function $f \in \text{PF}$ that is a $\leq_{\text{m}}^{\text{P}}$-reduction of A to some language B. Note that f is a $\leq_{\text{m}}^{\text{P}}$-reduction of A if and only if f is a $\leq_{\text{m}}^{\text{P}}$-reduction of A to $f(A) = \{f(x) \mid x \in A\}$. We say that A is *polynomial time, many-one reducible* (briefly, $\leq_{\text{m}}^{\text{P}}$-*reducible*) to B, and we write $A \leq_{\text{m}}^{\text{P}} B$, if there exists a $\leq_{\text{m}}^{\text{P}}$-reduction f of A to B. In this case, we also say that $A \leq_{\text{m}}^{\text{P}} B$ via f.

A language H is $\leq_{\text{m}}^{\text{P}}$-*hard* for a class $\mathcal{C}$ of languages if $A \leq_{\text{m}}^{\text{P}} H$ for all $A \in \mathcal{C}$. A language C is $\leq_{\text{m}}^{\text{P}}$-*complete* for $\mathcal{C}$ if $C \in \mathcal{C}$ and C is $\leq_{\text{m}}^{\text{P}}$-hard for $\mathcal{C}$. If $\mathcal{C} = \text{NP}$, this is the usual notion of NP-completeness [13]. In this paper we are especially concerned with languages that are $\leq_{\text{m}}^{\text{P}}$-hard or $\leq_{\text{m}}^{\text{P}}$-complete for E or E_2.

3. Resource-bounded measure. Resource-bounded measure [17], [21] is a very general theory whose special cases include classical Lebesgue measure, the measure structure of the class REC of all recursive languages, and measure in various complexity classes. In this paper, we are interested only in measure in E and E_2, so our discussion of measure is specific to these classes. The interested reader may consult §3 of [17] for more discussion and examples.

Throughout this section, we identify every language $A \subseteq \{0,1\}^*$ with its characteristic sequence $\chi_A \in \{0,1\}^\infty$, defined by $\chi_A[i] = [\![s_i \in A]\!]$ for all $i \in \mathbf{N}$. (Recall from §2 that

$s_0, s_1, s_2, \ldots$ is the standard enumeration of $\{0,1\}^*$.) We say that $x \in \{0,1\}^*$ is a *prefix*, or *partial specification*, of $A \subseteq \{0,1\}^*$ if x is a prefix of χ_A, i.e., if there exists $y \in \{0,1\}^\infty$ such that $\chi_A = xy$. In this case, we write $x \sqsubseteq A$. The set of all languages A for which x is a partial specification,

$$\mathbf{C}_x = \{A \subseteq \{0,1\}^* \mid x \sqsubseteq A\},$$

is the *cylinder specified by* the string $x \in \{0,1\}^*$. We say that the *measure* of the set $\mathbf{C}_x$ is $2^{-|x|}$. (Note that this is the probability that $A \in \mathbf{C}_x$ if $A \subseteq \{0,1\}^*$ is chosen probabilistically according to the random experiment in which an independent toss of a fair coin is used to decide membership of each string $x \in \{0,1\}^*$ in A.)

Notation. The classes $\mathrm{p}_1 = \mathrm{p}$ and p_2, both consisting of functions $f : \{0,1\}^* \to \{0,1\}^*$, are defined as follows.

$$\mathrm{p}_1 = \mathrm{p} = \{f \mid f \text{ is computable in polynomial time}\},$$
$$\mathrm{p}_2 = \{f \mid f \text{ is computable in } n^{(\log n)^{O(1)}} \text{ time}\}.$$

The measure structures of E and E_2 are developed in terms of the classes p_i, for $i = 1, 2$.

DEFINITION. A *density function* is a function $d : \{0,1\}^* \to [0, \infty)$ that satisfies

$$d(w) \geq \frac{d(w0) + d(w1)}{2} \tag{3.1}$$

for all $w \in \{0,1\}^*$. The *global value* of a density function d is $d(\lambda)$. The *set covered by* a density function d is

$$S[d] = \bigcup_{\substack{w \in \{0,1\}^* \\ d(w) \geq 1}} \mathbf{C}_w. \tag{3.2}$$

A density function d *covers* a set $X \subseteq \{0,1\}^\infty$ if $X \subseteq S[d]$.

For all density functions in this paper, equality actually holds in (3.1) above, but this is not required. Consider the random experiment in which a language $A \subseteq \{0,1\}^*$ is chosen by using an independent toss of a fair coin to decide whether each string $x \in \{0,1\}^*$ is in A. Taken together, parts (3.1) and (3.2) of the above definition imply that $\Pr[A \in S[d]] \leq d(\lambda)$ in this experiment. Intuitively, we regard a density function d as a "detailed verification" that $\Pr[A \in X] \leq d(\lambda)$ for all sets $X \subseteq S[d]$.

More generally, we are interested in "uniform systems" of density functions that are computable within some resource bound.

Since density functions are real-valued, their computations must employ finite approximations of real numbers. For this purpose, let

$$\mathbf{D} = \{m2^{-n} \mid m \in \mathbf{Z}, n \in \mathbf{N}\}$$

be the set of *dyadic rationals*. (These are rational numbers with finite binary expansions.) In order to have uniform criteria for computational complexity, we consider all functions of the form $f : X \to Y$, where each of the sets X, Y is $\mathbf{N}$, $\{0,1\}^*$, $\mathbf{D}$, or some Cartesian product of these sets, to really map $\{0,1\}^*$ into $\{0,1\}^*$. For example, a function $f : \mathbf{N}^2 \times \{0,1\}^* \to \mathbf{N} \times \mathbf{D}$ is formally interpreted as a function $\tilde{f} : \{0,1\}^* \to \{0,1\}^*$. Under this interpretation, $f(i, j, w) = (k, q)$ means that $\tilde{f}(\langle 0^i, \langle 0^j, w\rangle\rangle) = \langle 0^k, \langle u, v\rangle\rangle$, where u and v are the binary representations of the integer and fractional parts of q, respectively. Moreover, we only care about the values of $\tilde{f}$ for arguments of the form $\langle 0^i, \langle 0^j, w\rangle\rangle$, and we insist that these values have the form $\langle 0^k, \langle u, v\rangle\rangle$ for such arguments.

DEFINITION. An n-dimensional *density system* (n-*DS*) is a function

$$d : \mathbf{N}^n \times \{0,1\}^* \to [0,\infty)$$

such that $d_{\vec{k}}$ is a density function for every $\vec{k} \in \mathbf{N}^n$. It is sometimes convenient to regard a density function as a 0-DS.

DEFINITION. A *computation* of an n-DS d is a function $\hat{d} : \mathbf{N}^{n+1} \times \{0,1\}^* \to \mathbf{D}$ such that

$$|\hat{d}_{\vec{k},r}(w) - d_{\vec{k}}(w)| \leq 2^{-r}$$

for all $\vec{k} \in \mathbf{N}^n$, $r \in \mathbf{N}$, and $w \in \{0,1\}^*$. For $i = 1, 2$, a p_i-*computation* of an n-DS d is a computation $\hat{d}$ of d such that $\hat{d} \in \mathrm{p}_i$. An n-DS d is p_i-*computable* if there exists a p_i-computation $\hat{d}$ of d.

If d is an n-DS such that $d : \mathbf{N}^n \times \{0,1\}^* \to \mathbf{D}$ and $d \in \mathrm{p}_i$, then d is trivially p_i-computable. This fortunate circumstance, in which there is no need to compute approximations, frequently occurs in practice. (Such applications typically involve approximations, but these are "hidden" by invoking fundamental theorems whose proofs involve approximations).

We now come to the key idea of resource-bounded measure theory.

DEFINITION. A *null cover* of a set $X \subseteq \{0,1\}^\infty$ is a 1-DS d such that, for all $k \in \mathbf{N}$, d_k covers X with global value $d_k(\lambda) \leq 2^{-k}$. For $i = 1, 2$, a p_i-*null cover* of X is a null cover of X that is p_i-computable.

In other words, a null cover of X is a uniform system of density functions that cover X with rapidly vanishing global value. It is easy to show that a set $X \subseteq \{0,1\}^\infty$ has classical Lebesgue measure 0 (i.e., probability 0 in the above coin-tossing experiment) if and only if there exists a null cover of X.

DEFINITION. A set X has p_i-*measure* 0, and we write $\mu_{\mathrm{p}_i}(X) = 0$ if there exists a p_i-null cover of X. A set X has p_i-*measure* 1, and we write $\mu_{\mathrm{p}_i}(X) = 1$ if $\mu_{\mathrm{p}_i}(X^c) = 0$.

Thus a set X has p_i-measure 0 if p_i provides sufficient computational resources to compute uniformly good approximations to a system of density functions that cover X with rapidly vanishing global value.

We now turn to the internal measure structures of the classes $\mathrm{E} = \mathrm{E}_1 = \mathrm{DTIME}(2^{\mathrm{linear}})$ and $\mathrm{E}_2 = \mathrm{DTIME}(2^{\mathrm{polynomial}})$.

DEFINITION. A set X has *measure* 0 *in* E_i, and we write $\mu(X \mid \mathrm{E}_i) = 0$, if $\mu_{\mathrm{p}_i}(X \cap \mathrm{E}_i) = 0$. A set X has *measure* 1 *in* E_i, and we write $\mu(X \mid \mathrm{E}_i) = 1$, if $\mu(X^c \mid \mathrm{E}_i) = 0$. If $\mu(X \mid \mathrm{E}_i) = 1$, we say that *almost every* language in E_i is in X.

We write $\mu(X|\mathrm{E}_i) \neq 0$ to indicate that X does *not* have measure 0 in E_i. Note that this does *not* assert that "$\mu(X|\mathrm{E}_i)$" has some nonzero value.

The following is obvious but useful.

FACT 3.1. *For every set* $X \subseteq \{0,1\}^\infty$,

$$\begin{array}{ccccc} \mu_{\mathrm{p}}(X) = 0 & \Longrightarrow & \mu_{\mathrm{p}_2}(X) = 0 & \Longrightarrow & \Pr[A \in X] = 0 \\ \Downarrow & & \Downarrow & & \\ \mu(X|\mathrm{E}) = 0 & & \mu(X|\mathrm{E}_2) = 0, & & \end{array}$$

where the probability $\Pr[A \in X]$ *is computed according to the random experiment in which a language* $A \subseteq \{0,1\}^*$ *is chosen probabilistically by using an independent toss of a fair coin to decide whether each string* $x \in \{0,1\}^*$ *is in* A.

It is shown in [17] that these definitions endow E and E_2 with internal measure structure. This structure justifies the intuition that if $\mu(X|\mathrm{E}) = 0$, then $X \cap \mathrm{E}$ is a *negligibly small* subset of E (and similarly for E_2). The next two results state aspects of this structure that are especially relevant to the present work.

THEOREM 3.2 [17]. *For all cylinders* $\mathbf{C}_w$, $\mu(\mathbf{C}_w|\mathrm{E}) \neq 0$ *and* $\mu(\mathbf{C}_w|\mathrm{E}_2) \neq 0$. *In particular,* $\mu(\mathrm{E}|\mathrm{E}) \neq 0$ *and* $\mu(\mathrm{E}_2|\mathrm{E}_2) \neq 0$.

The next lemma, which is used in proving Theorem 4.6 and Lemma 5.16, involves the following computational restriction of the notion of "countable union."

DEFINITION. Let $i \in \{1, 2\}$ and let $Z, Z_0, Z_1, Z_2, \cdots \subseteq \{0, 1\}^\infty$. Then Z is a p_i-*union of the* p_i-*measure* 0 *sets* $Z_0, Z_1, Z_2, \cdots$ if $Z = \cup_{j=0}^{\infty} Z_j$ and there exists a p_i-computable 2-DS d such that each d_j is a p_i-null cover of Z_j.

LEMMA 3.3 [17]. *Let* $i \in \{1, 2\}$ *and let* $Z, Z_0, Z_1, Z_2, \ldots \subseteq \{0, 1\}^\infty$. *If* Z *is a* p_i-*union of the* p_i-*measure* 0 *sets* $Z_0, Z_1, Z_2, \ldots$, *then* Z *has* p_i-*measure* 0. □

4. Complexity cores: Lower bounds. Orponen and Schöning [28] have shown that every $\leq_{\mathrm{m}}^{\mathrm{P}}$-hard language for E has a dense polynomial complexity core. In this section we extend this result by proving that every weakly $\leq_{\mathrm{m}}^{\mathrm{P}}$-hard language for E has a dense exponential complexity core. We begin by explaining our terminology.

Given a machine M and an input $x \in \{0, 1\}^*$, we write $M(x) = 1$ if M accepts x, $M(x) = 0$ if M rejects x, and $M(x) = \perp$ in any other case (i.e., if M fails to halt or M halts without deciding x). If $M(x) \in \{0, 1\}$, we write $\mathrm{time}_M(x)$ for the number of steps used in the computation of $M(x)$. If $M(x) = \perp$, we define $\mathrm{time}_M(x) = \infty$. We partially order the set $\{0, 1, \perp\}$ by $\perp < 0$ and $\perp < 1$, with 0 and 1 incomparable. A machine M is *consistent* with a language $A \subseteq \{0, 1\}^*$ if $M(x) \leq [\![x \in A]\!]$ for all $x \in \{0, 1\}^*$.

DEFINITION. Let $t : \mathbf{N} \to \mathbf{N}$ be a time bound and let $A, K \subseteq \{0, 1\}^*$. Then K is a DTIME($t(n)$)-*complexity core* of A if, for every $c \in \mathbf{N}$ and every machine M that is consistent with A, the "fast set"

$$F = \{x \,|\, \mathrm{time}_M(x) \leq c \cdot t(|x|) + c\}$$

satisfies $|F \cap K| < \infty$. (By our definition of $\mathrm{time}_M(x)$, $M(x) \in \{0, 1\}$ for all $x \in F$. Thus F is the set of all strings that M decides efficiently.)

Remark. The previous definition quantifies over all machines consistent with A, while the standard definition of complexity cores (cf. [3]) quantifies only over machines that *decide* A. For recursive languages A (and time-constructible bounds t), it is easy to see that the above definition is exactly equivalent to the standard definition. However, the above definition is stronger than the standard definition when A is not recursive. For example, consider *tally* languages (i.e., languages $A \subseteq \{0\}^*$). Under our definition, every DTIME(n)-complexity core K of every tally language must satisfy $|K - \{0\}^*| < \infty$. However, under the standard definition, complexity cores are only defined for recursive sets A (as in [3]), or else *every* set $K \subseteq \{0, 1\}^*$ is *vacuously* a complexity core for *every* nonrecursive language (tally or otherwise). Thus by quantifying over all machines consistent with A, our definition makes the notion of complexity core meaningful for nonrecursive languages A. This enables one to eliminate the extraneous hypothesis that A is recursive from several results. In some cases, this improvement is of little interest. However, in §6, we show that *every* $\leq_{\mathrm{m}}^{\mathrm{P}}$-hard language H for E has unusually small complexity cores. This upper bound holds regardless of whether H is recursive.

Note that every subset of a DTIME($t(n)$)-complexity core of A is a DTIME($t(n)$)-complexity core of A. Note also that, if $s(n) = O(t(n))$, then every DTIME($t(n)$)-complexity core of A is a DTIME($s(n)$)-complexity core of A.

DEFINITION. Let $A, K \subseteq \{0, 1\}^*$.

1. K is a *polynomial complexity core* (or, briefly, a P-*complexity core*) of A if K is a DTIME(n^k)-complexity core of A for all $k \in \mathbf{N}$.

2. K is an *exponential complexity core* of A if there is a real number $\epsilon > 0$ such that K is a DTIME(2^{n^ϵ})-complexity core of A.

Much of our work here uses languages that are "incompressible by many-one reductions," an idea originally exploited by Meyer [26]. The following definitions develop this notion.

DEFINITION. The *collision set* of a function $f : \{0,1\}^* \to \{0,1\}^*$ is

$$C_f = \{x \in \{0,1\}^* \mid (\exists y < x) f(y) = f(x)\}.$$

Here, we are using the standard ordering $s_0 < s_1 < s_2 < \cdots$ of $\{0,1\}^*$.

Note that f is one-to-one if and only if $C_f = \emptyset$.

DEFINITION. A function $f : \{0,1\}^* \to \{0,1\}^*$ is *one-to-one almost everywhere* (or, briefly, *one-to-one a.e.*) if its collision set C_f is finite.

DEFINITION. Let $A, B \subseteq \{0,1\}^*$ and let $t : \mathbf{N} \to \mathbf{N}$. A $\leq_m^{\mathrm{DTIME}(t)}$*-reduction of* A *to* B is a function $f \in \mathrm{DTIMEF}(t)$ such that $A = f^{-1}(B)$, i.e., such that, for all $x \in \{0,1\}^*$, $x \in A$ iff $f(x) \in B$. A $\leq_m^{\mathrm{DTIME}(t)}$*-reduction of* A is a function f that is a $\leq_m^{\mathrm{DTIME}(t)}$-reduction of A to $f(A)$.

It is easy to see that f is a $\leq_m^{\mathrm{DTIME}(t)}$-reduction of A if and only if there exists a language B such that f is a $\leq_m^{\mathrm{DTIME}(t)}$-reduction of A to B.

DEFINITION. Let $t : \mathbf{N} \to \mathbf{N}$. A language $A \subseteq \{0,1\}^*$ is *incompressible by* $\leq_m^{\mathrm{DTIME}(t)}$*-reductions* if every $\leq_m^{\mathrm{DTIME}(t)}$-reduction of A is one-to-one a.e. A language $A \subseteq \{0,1\}^*$ is *incompressible by* $\leq_m^{\mathrm{P}}$*-reductions* if it is incompressible by $\leq_m^{\mathrm{DTIME}(q)}$-reductions for all polynomials q.

Intuitively, if f is a $\leq_m^{\mathrm{DTIME}(t)}$-reduction of A to B and C_f is large, then f compresses many questions "$x \in A$?" to fewer questions "$f(x) \in B$?" If A is incompressible by $\leq_m^{\mathrm{P}}$-reductions, then very little such compression can occur.

Our first observation, an obvious generalization of a result of Balcázar and Schöning [4] (see Corollary 4.5), relates incompressibility to complexity cores.

LEMMA 4.4. *If* $t : \mathbf{N} \to \mathbf{N}$ *is time constructible, then every language that is incompressible by* $\leq_m^{\mathrm{DTIME}(t)}$*-reductions has* $\{0,1\}^*$ *as a DTIME*(t)*-complexity core.*

Proof. Let A be a language that does not have $\{0,1\}^*$ as a DTIME(t)-complexity core. It suffices to prove that A is not incompressible by $\leq_m^{\mathrm{DTIME}(t)}$-reductions. This is clear if $A = \emptyset$ or $A = \{0,1\}^*$, so assume that $\emptyset \neq A \neq \{0,1\}^*$. Fix $u \in A$ and $v \in A^c$. Since $\{0,1\}^*$ is not a DTIME(t)-complexity core of A, there exist $c \in \mathbf{N}$ and a machine M such that M is consistent with A and the fast set

$$F = \{x \mid \mathrm{time}_M(x) \leq c \cdot t(|x|) + c\}$$

is infinite. Define a function $f : \{0,1\}^* \to \{0,1\}^*$ by

$$f(x) = \begin{cases} u & \text{if } M(x) = 1 \text{ in} \leq c \cdot t(|x|) + c \text{ steps,} \\ v & \text{if } M(x) = 0 \text{ in} \leq c \cdot t(|x|) + c \text{ steps,} \\ x & \text{otherwise.} \end{cases}$$

Since t is time-constructible, $f \in \mathrm{DTIMEF}(t)$. Since M is consistent with A, f is a $\leq_m^{\mathrm{DTIME}(t)}$-reduction of A to A. Since F is infinite, at least one of the sets $f^{-1}(\{u\})$, $f^{-1}(\{v\})$ is infinite, so the collision set C_f is infinite. Thus A is not incompressible by $\leq_m^{\mathrm{DTIME}(t)}$-reductions. □

COROLLARY 4.5. *Let* $c \in \mathbf{N}$.

1 (*Balcazar and Schöning* [4]). *Every language that is incompressible by* $\leq_m^{\mathrm{P}}$*-reductions has* $\{0,1\}^*$ *as a P-complexity core.*

2. *Every language that is incompressible by* $\leq_m^{\mathrm{DTIME}(2^{cn})}$*-reductions has* $\{0,1\}^*$ *as a DTIME*(2^{cn})*-complexity core.*

3. *Every language that is incompressible by* $\leq_m^{\mathrm{DTIME}(2^{n^c})}$*-reductions has* $\{0,1\}^*$ *as a DTIME*(2^{n^c})*-complexity core.* □

We now prove that, in E and E_2, almost every language is incompressible by $\leq_m^{DTIME(t)}$-reductions, for exponential time bounds t.

THEOREM 4.6. *Let $c \in \mathbf{Z}^+$ and define the sets*

$$X = \{A \subseteq \{0,1\}^* | A \text{ is incompressible by } \leq_m^{DTIME(2^{cn})}\text{-reductions}\},$$

$$Y = \{A \subseteq \{0,1\}^* | A \text{ is incompressible by } \leq_m^{DTIME(2^{n^c})}\text{-reductions}\}.$$

Then $\mu_p(X) = \mu_{p_2}(Y) = 1$. Thus almost every language in E *is incompressible by $\leq_m^{DTIME(2^{cn})}$-reductions, and almost every language in* E_2 *is incompressible by $\leq_m^{DTIME(2^{n^c})}$-reductions.*

Proof. Let $c \in \mathbf{Z}^+$. We prove that $\mu_p(X) = 1$. The proof that $\mu_{p_2}(X) = 1$ is analogous.

Let $f \in DTIMEF(2^{(c+1)n})$ be a function that is universal for $DTIMEF(2^{cn})$, in the sense that

$$DTIMEF(2^{cn}) = \{f_i \mid i \in \mathbf{N}\}.$$

For each $i \in \mathbf{N}$, define a set Z_i of languages as follows: If the collision set C_{f_i} is finite, then $Z_i = \emptyset$. Otherwise, if C_{f_i} is infinite, then Z_i is the set of all languages A such that f_i is a $\leq_m^{DTIME(2^{cn})}$-reduction of A.

Define a function $d : \mathbf{N} \times \mathbf{N} \times \{0,1\}^* \to [0, \infty)$ as follows: Let $i, k \in \mathbf{N}$ be arbitrary, let $w \in \{0,1\}^*$, and let $b \in \{0,1\}$.

(i) $d_{i,k}(\lambda) = 2^{-k}$.

(ii) If $s_{|w|} \notin C_{f_i}$, then $d_{i,k}(wb) = d_{i,k}(w)$.

(iii) If $s_{|w|} \in C_{f_i}$, then fix the least $j \in \mathbf{N}$ such that $f_i(s_j) = f_i(s_{|w|})$ and set

$$d_{i,k}(wb) = 2 \cdot d_{i,k}(w) \cdot [\![b = w[j]]\!].$$

It is clear that d is a 2-DS. Since $f \in DTIMEF(2^{(c+1)n})$ and the computation of $d_{i,k}(w)$ only uses values $f_i(u)$ for strings u with $|u| = O(\log |w|)$, it is also clear that $d \in p$, so d is a p-computable 2-DS.

We now show that $Z_i \subseteq S[d_{i,k}]$ for all $i, k \in \mathbf{N}$. If C_{f_i} is finite, then this is clear (because $Z_i = \emptyset$), so assume that C_{f_i} is infinite and let $A \in Z_i$. Let w be a string consisting of the first l bits of the characteristic sequence of A, where s_{l-1} is the kth element of C_{f_i}. This choice of l ensures that clause (iii) of the definition of d is invoked exactly k times in the recursive computation of $d_{i,k}(w)$. Since f_i is a $\leq_m^{DTIME(2^{cn})}$-reduction of A (because $A \in Z_i$), we have $b = w[j]$ in each of these k invocations, so

$$d_{i,k}(w) = 2^k \cdot d_{i,k}(\lambda) = 1.$$

Thus $A \in \mathbf{C}_w \subseteq S[d_{i,k}]$. This confirms that $Z_i \subseteq S[d_{i,k}]$ for all $i, k \in \mathbf{N}$. It follows easily that for each $i \in \mathbf{N}$, d_i is a p-null cover of Z_i. This implies that

$$X^c = \bigcup_{k=0}^{\infty} Z_k$$

is a p-union of p-measure 0 sets, whence $\mu_p(X) = 1$ by Lemma 3.3. □

COROLLARY 4.7. *Almost every language in* E *and almost every language in* E_2 *is incompressible by $\leq_m^P$-reductions.* □

COROLLARY 4.8 (Meyer [26]). *There is a language $A \in$ E that is incompressible by $\leq_m^P$-reductions.* □

COROLLARY 4.9. *Let $c \in \mathbf{Z}^+$.*

1. *Almost every language in* E *has* $\{0, 1\}^*$ *as a DTIME*(2^{cn})*-complexity core.*

2. *Almost every language in* E_2 *has* $\{0, 1\}^*$ *as a DTIME*(2^{n^c})*-complexity core.* □

We now consider complexity cores of $\leq_m^P$-hard languages. Our starting point is the following two known facts.

FACT 4.10 (Orponen and Schöning [28]). *Every language that is* $\leq_m^P$*-hard for* E (*equivalently, for* E_2) *has a dense P-complexity core.*

FACT 4.11 (Orponen and Schöning [28]). *If* $P \neq NP$, *then every language that is* $\leq_m^P$*-hard for* NP *has a nonsparse P-complexity core.*

We first extend Fact 4.10. For this, we need a definition. The *lower* $\leq_m^P$*-span* of a language $A \subseteq \{0, 1\}^*$ is

$$P_m(A) = \{B \subseteq \{0, 1\}^* \mid B \leq_m^P A\},$$

i.e., the set of all languages lying at or below A in the $\leq_m^P$-reducibility structure of the set of all languages. Recall that a language A is $\leq_m^P$*-hard* for a complexity class $\mathcal{C}$ if $\mathcal{C} \subseteq P_m(A)$.

DEFINITION. A language $A \subseteq \{0, 1\}^*$ is *weakly* $\leq_m^P$*-hard* for E (respectively, for E_2) if $\mu(P_m(A) \mid E) \neq 0$ (respectively, $\mu(P_m(A) \mid E_2) \neq 0$). A language $A \subseteq \{0, 1\}^*$ is *weakly* $\leq_m^P$*-complete* for E (respectively, for E_2) if $A \in E$ (respectively, $A \in E_2$) and A is weakly $\leq_m^P$-hard for E (respectively, for E_2).

Thus a language A is weakly $\leq_m^P$-hard for E if a nonnegligible subset of the languages in E are $\leq_m^P$-reducible to A. Very recently, Lutz [19] has established the existence of languages that are weakly $\leq_m^P$-complete, but not $\leq_m^P$-complete, for E (and similarly for E_2). Although "$\leq_m^P$-hard for E" and "$\leq_m^P$-hard for E_2" are equivalent, we do not know the relationship between "weakly $\leq_m^P$-hard for E" and "weakly $\leq_m^P$-hard for E_2."

Recall that a language $D \subseteq \{0, 1\}^*$ is *dense* if there is a real number $\epsilon > 0$ such that $|D_{\leq n}| > 2^{n^\epsilon}$ a.e.

THEOREM 4.12. *Every language that is weakly* $\leq_m^P$*-hard for* E *or* E_2 *has a dense exponential complexity core.*

Proof. We prove this for E. The proof for E_2 is identical.

Let H be a language that is weakly $\leq_m^P$-hard for E. Then $P_m(H)$ does not have measure 0 in E, so by Theorem 4.6, there is a language $A \in P_m(H)$ that is incompressible by $\leq_m^{DTIME(2^n)}$-reductions. Let f be a $\leq_m^P$-reduction of A to H, let q be a strictly increasing polynomial bound on the time required to compute f, and let $\epsilon = \frac{1}{3 \cdot \deg(q)}$. Then the language $K = f(\{0, 1\}^*)$ is a dense DTIME(2^{n^ϵ})-complexity core of H. □

Lutz has proposed the investigation of the consequences of the strong hypotheses $\mu(NP \mid E) \neq 0$ and $\mu(NP \mid E_2) \neq 0$ [18], [23], [22]. In this regard, we have the following.

COROLLARY 4.13. *If* $\mu(NP \mid E) \neq 0$ *or* $\mu(NP \mid E_2) \neq 0$, *then every* $\leq_m^P$*-hard language for* NP *has a dense exponential complexity core.* □

Thus, for example, if NP is not small, then there is a dense set K of Boolean formulas in conjunctive normal form such that every machine that is consistent with SAT performs exponentially badly (either by running for more than $2^{|x|^\epsilon}$ steps or by failing to decide) on all but finitely many inputs $x \in K$.

Note that Theorem 4.12 extends Fact 4.10 and that Corollary 4.13 has a stronger hypothesis and stronger conclusion than Fact 4.11. Note also that Corollary 4.13 holds with NP replaced by PH, PP, PSPACE, or any class whatsoever.

The following result shows that the density bounds of Theorem 4.12 and Corollary 4.13 are tight.

THEOREM 4.14. *For every* $\epsilon > 0$, *each of the classes* NP, E, *and* E_2 *has a* $\leq_m^P$*-complete language, every P-complexity core* K *of which satisfies* $|K_{\leq n}| < 2^{n^\epsilon}$ *a.e.*

Proof. Let $\epsilon > 0$, let $\mathcal{C}$ be any one of the classes NP, E, E_2, and let A be a language that is $\leq_m^P$-complete for $\mathcal{C}$. Let $k = \lceil \frac{2}{\epsilon} \rceil$ and define the language

$$B = \{x10^{|x|^k} \mid x \in A\}.$$

Then B is $\leq_m^P$-complete for $\mathcal{C}$ and every P-complexity core K of B satisfies $|K_{\leq n}| < 2^{n^\epsilon}$ a.e. □

5. Measure of degrees. In this section we prove that all $\leq_m^P$-degrees have measure 0 in the complexity classes E and E_2. This fact and more follow from the Small Span Theorem, which we prove first.

Recall that the *lower* $\leq_m^P$*-span* of a language $A \subseteq \{0,1\}^*$ is

$$P_m(A) = \{B \subseteq \{0,1\}^* \mid B \leq_m^P A\}.$$

Similarly, define the *upper* $\leq_m^P$*-span* of A to be

$$P_m^{-1}(A) = \{B \subseteq \{0,1\}^* \mid A \leq_m^P B\}.$$

The $\leq_m^P$*-degree* of A is then

$$\deg_m^P(A) = P_m(A) \cap P_m^{-1}(A),$$

the intersection of the upper and lower spans.

The main result of this section is that, if A is in E or E_2, then at least one of the spans $P_m(A)$, $P_m^{-1}(A)$ is small.

THEOREM 5.15 (Small Span Theorem).

1. *For every* $A \in E$,

$$\mu(P_m(A) \mid E) = 0$$

or

$$\mu_p(P_m^{-1}(A)) = \mu(P_m^{-1}(A) \mid E) = 0.$$

2. *For every* $A \in E_2$,

$$\mu(P_m(A) \mid E_2) = 0$$

or

$$\mu_{p_2}(P_m^{-1}(A)) = \mu(P_m^{-1}(A) \mid E_2) = 0.$$

We first use the following lemma to prove Theorem 5.15. We then prove the lemma.

LEMMA 5.16. *Let A be a language that is incompressible by $\leq_m^P$-reductions.*

1. *If* $A \in E$, *then* $\mu_p(P_m^{-1}(A)) = \mu(P_m^{-1}(A)|E) = 0$.

2. *If* $A \in E_2$, *then* $\mu_{p_2}(P_m^{-1}(A)) = \mu(P_m^{-1}(A)|E_2) = 0$.

Proof of Theorem 5.15. To prove 1, let $A \in E$ and let X be the set of all languages that are incompressible by $\leq_m^P$-reductions. We have two cases.

Case I. If $P_m(A) \cap E \cap X = \emptyset$, then Corollary 4.7 tells us that $\mu(P_m(A) \mid E) = 0$.

Case II. If $P_m(A) \cap E \cap X \neq \emptyset$, then fix a language $B \in P_m(A) \cap E \cap X$. Since $B \in E \cap X$, Lemma 5.16 tells us that

$$\mu_p(P_m^{-1}(B)) = \mu(P_m^{-1}(B) \mid E) = 0.$$

Since $P_m^{-1}(A) \subseteq P_m^{-1}(B)$, it follows that

$$\mu_p(P_m^{-1}(A)) = \mu(P_m^{-1}(A) \mid E) = 0.$$

This proves 1. The proof of 2 is identical. □

Proof of Lemma 5.16. To prove 1, let $A \in E$ be incompressible by $\leq_m^P$-reductions. Let $f \in DTIMEF(2^n)$ be a function that is universal for PF, in the sense that

$$PF = \{f_i \mid i \in \mathbf{N}\}.$$

For each $i \in \mathbf{N}$, define the set Z_i of languages as follows. If the collision set C_{f_i} is infinite, then $Z_i = \emptyset$. Otherwise, if C_{f_i} is finite, then

$$Z_i = \{B \subseteq \{0,1\}^* \mid A \leq_m^P B \text{ via } f_i\}.$$

Note that

$$P_m^{-1}(A) = \bigcup_{i=0}^{\infty} Z_i,$$

because A is incompressible by $\leq_m^P$-reductions.

Define a function $d : \mathbf{N} \times \mathbf{N} \times \{0,1\}^* \to [0, \infty)$ as follows. Let $i, k \in \mathbf{N}$ be arbitrary, let $w \in \{0,1\}^*$, and let $b \in \{0,1\}$.

(i) $d_{i,k}(\lambda) = 2^{-k}$.

(ii) If there is no $j \leq 2|w|$ such that $f_i(s_j) = s_{|w|}$, then $d_{i,k}(wb) = d_{i,k}(w)$.

(iii) If there exists $j \leq 2|w|$ such that $f_i(s_j) = s_{|w|}$, then fix the least such j and set

$$d_{i,k}(wb) = 2 \cdot d_{i,k}(w) \cdot [\![b = [\![s_j \in A]\!]]\!].$$

It is clear that d is a 2-DS. Also, since $f \in DTIMEF(2^n)$ and $A \in E$, it is easy to see that $d \in p$, whence d is a p-computable 2-DS.

We now show that $Z_i \subseteq S[d_{i,k}]$ for all $i, k \in \mathbf{N}$. If C_{f_i} is infinite, then this is clear (because $Z_i = \emptyset$), so assume that $|C_{f_i}| = c < \infty$ and let $B \in Z_i$, i.e., $A \leq_m^P B$ via f_i. Let v be the string consisting of the first l bits of the characteristic sequence of B, where l is large enough that

$$f_i(\{s_0, \ldots, s_{2k+4c-1}\}) \subseteq \{s_0, \ldots, s_{l-1}\}.$$

Consider the computation of $d_{i,k}(v)$ by clauses (i), (ii), and (iii). Since $A \leq_m^P B$ via f_i, clause (iii) does not cause $d_{i,k}(w)$ to be 0 for any prefix w of v. Let

$$S = \{s_n \mid 0 \leq n < 2k + 4c \text{ and } f_i(s_n) \notin \{s_0, \ldots, s_{\lceil \frac{n}{2} \rceil - 1}\}\}$$

and

$$T = f_i(S),$$

then clause (iii) doubles the density whenever $s_{|w|} \in T$, so

$$d_{i,k}(v) \geq 2^{|T|} d_{i,k}(\lambda) = 2^{|T|-k} \geq 2^{|S|-k-c}.$$

Also, if

$$S' = \{s_n \mid 0 \leq n < 2k + 4c \text{ and } f_i(s_n) \notin \{s_0, \ldots, s_{k+2c-1}\}\},$$

then $S' \subseteq S$ and

$$|S'| \geq (2k + 4c) - (k + 2c) - c = k + c.$$

Putting this all together, we have

$$d_{i,k}(v) \geq 2^{|S|-k-c} \geq 2^{|S'|-k-c} \geq 1,$$

whence $B \in \mathbf{C}_v \subseteq S[d_{i,k}]$. This shows that $Z_i \subseteq S[d_{i,k}]$ for all $i, k \in \mathbf{N}$.

Since d is p-computable and $d_{i,k}(\lambda) = 2^{-k}$ for all $i, k \in \mathbf{N}$, it follows that, for all $i \in \mathbf{N}$, d_i is p-null cover of Z_i. This implies that $\mathrm{P}_{\mathrm{m}}^{-1}(A)$ is a p-union of the p-measure 0 sets Z_i. It follows by Lemma 3.3 that $\mu_{\mathrm{p}}(\mathrm{P}_{\mathrm{m}}^{-1}(A)) = \mu(\mathrm{P}_{\mathrm{m}}^{-1}(A) \mid \mathrm{E}) = 0$. This completes the proof of 1.

The proof of 2 is identical. One need only note that, if $A \in \mathrm{E}_2$, then $d \in \mathrm{p}_2$. □

Remark. Ambos-Spies [1] has shown that $\mathrm{P}_{\mathrm{m}}(A)$ has Lebesgue measure 0 whenever $A \notin \mathrm{P}$. Lemma 5.16 obtains a stronger conclusion (resource-bounded measure 0) from a stronger hypothesis on A.

It is now straightforward to derive consequences of these results for the structure of E and E_2. We first note that $\leq_{\mathrm{m}}^{\mathrm{P}}$-hard languages for E are extremely rare.

THEOREM 5.17. *Let $\mathcal{H}_{\mathrm{E}}$ be the set of all languages that are $\leq_{\mathrm{m}}^{\mathrm{P}}$-hard for* E. *Then* $\mu_{\mathrm{p}}(\mathcal{H}_{\mathrm{E}}) = 0$.

Proof. Let A be as in Corollary 4.8. Then $\mathcal{H}_{\mathrm{E}} \subseteq \mathrm{P}_{\mathrm{m}}^{-1}(A)$, so Lemma 5.16 tells us that

$$\mu_{\mathrm{p}}(\mathcal{H}_{\mathrm{E}}) = \mu_{\mathrm{p}}(\mathrm{P}_{\mathrm{m}}^{-1}(A)) = 0. \qquad \square$$

Theorem 5.17 immediately yields an alternate proof of the following result.

COROLLARY 5.18 (Mayordomo[25]). *Let $\mathcal{C}_{\mathrm{E}}$, $\mathcal{C}_{\mathrm{E}_2}$ be the sets of languages that are $\leq_{\mathrm{m}}^{\mathrm{P}}$-complete for* E, E_2, *respectively. Then* $\mu(\mathcal{C}_{\mathrm{E}} | \mathrm{E}) = \mu(\mathcal{C}_{\mathrm{E}_2} | \mathrm{E}_2) = 0$. □

(Mayordomo's proof of Corollary 5.18 used Berman's result [6], that no $\leq_{\mathrm{m}}^{\mathrm{P}}$-complete language for E is P-immune.)

As it turns out, Corollary 5.18 is only a special case of the following general result. *All* $\leq_{\mathrm{m}}^{\mathrm{P}}$-degrees have measure 0 in E and in E_2.

THEOREM 5.19. *For all* $A \subseteq \{0, 1\}^*$,

$$\mu(\deg_{\mathrm{m}}^{\mathrm{P}}(A) \mid \mathrm{E}) = \mu(\deg_{\mathrm{m}}^{\mathrm{P}}(A) \mid \mathrm{E}_2) = 0.$$

Proof. Let $A \subseteq \{0, 1\}^*$. We prove that $\mu(\deg_{\mathrm{m}}^{\mathrm{P}}(A) \mid \mathrm{E}) = 0$. The proof that $\mu(\deg_{\mathrm{m}}^{\mathrm{P}}(A) \mid \mathrm{E}_2) = 0$ is identical (in fact simpler, because E_2 is closed under $\leq_{\mathrm{m}}^{\mathrm{P}}$).

If $\deg_{\mathrm{m}}^{\mathrm{P}}(A) \cap \mathrm{E} = \emptyset$, then $\mu(\deg_{\mathrm{m}}^{\mathrm{P}}(A) \mid \mathrm{E}) = 0$ holds trivially, so assume that $\deg_{\mathrm{m}}^{\mathrm{P}}(A) \cap \mathrm{E} \neq \emptyset$. Fix $B \in \deg_{\mathrm{m}}^{\mathrm{P}}(A) \cap \mathrm{E}$. Then, by Theorem 5.15,

$$\mu(\deg_{\mathrm{m}}^{\mathrm{P}}(B) \mid \mathrm{E}) = \mu(\mathrm{P}_{\mathrm{m}}(B) \mid \mathrm{E}) = 0$$

or

$$\mu(\deg_{\mathrm{m}}^{\mathrm{P}}(B) \mid \mathrm{E}) = \mu(\mathrm{P}_{\mathrm{m}}^{-1}(B) \mid \mathrm{E}) = 0.$$

Since $\deg_{\mathrm{m}}^{\mathrm{P}}(A) = \deg_{\mathrm{m}}^{\mathrm{P}}(B)$, it follows that $\mu(\deg_{\mathrm{m}}^{\mathrm{P}}(A) \mid \mathrm{E}) = 0$. □

We now have the following two corollaries for NP.

COROLLARY 5.20. *Let $\mathcal{H}_{\mathrm{NP}}$ be the set of languages that are $\leq_{\mathrm{m}}^{\mathrm{P}}$-hard for* NP.

1. *If* $\mu(\mathrm{NP} \mid \mathrm{E}) \neq 0$, *then* $\mu(\mathcal{H}_{\mathrm{NP}} \mid \mathrm{E}) = 0$.
2. *If* $\mu(\mathrm{NP} \mid \mathrm{E}_2) \neq 0$, *then* $\mu(\mathcal{H}_{\mathrm{NP}} \mid \mathrm{E}_2) = 0$.

Proof. This follows immediately from Theorem 5.15, with $A = \mathrm{SAT}$. □

COROLLARY 5.21. *Let $\mathcal{C}_{\mathrm{NP}}$ be the set of languages that are $\leq_m^P$-complete for* NP. *Then* $\mu(\mathcal{C}_{\mathrm{NP}} \mid \mathrm{E}) = \mu(\mathcal{C}_{\mathrm{NP}} \mid \mathrm{E}_2) = 0$.

Proof. Since $\mathcal{C}_{\mathrm{NP}} = \deg_m^P(\mathrm{SAT})$, this follows immediately from Theorem 5.19. □

It is interesting to note that Corollary 5.21, unlike Corollary 5.20, is an absolute result, requiring no unproven hypothesis. The price we pay for this is that we do not know *why* it holds! For example, the Small Span Theorem tells us that $\mathcal{C}_{\mathrm{NP}} = \mathcal{H}_{\mathrm{NP}} \cap \mathrm{NP}$ has measure 0 in E because $\mu(\mathcal{H}_{\mathrm{NP}} \mid \mathrm{E}) = 0$ *or* $\mu(\mathrm{NP} \mid \mathrm{E}) = 0$, but it does *not* tell us which of these two very different situations occurs.

Note that Corollaries 5.20 and 5.21 also hold with NP replaced by *any other class whatsoever*.

We conclude this section by noting two respects in which the Small Span Theorem cannot be improved. First, the hypotheses $A \in \mathrm{E}$ and $A \in \mathrm{E}_2$ are essential for parts 1 and 2, respectively. For example, if A is p-random [20], then $\mu_p(\{A\}) \neq 0$, so none of $\deg_m^P(A)$, $P_m(A)$, $P_m^{-1}(A)$ can have p-measure 0.

The second respect in which the Small Span Theorem cannot be improved involves the variety of small-span configurations. In both E and E_2, either one or both of the upper and lower spans of a language can in fact be small. We give examples for E.

(a) It is well known [26] that there is a language $A \in \mathrm{E}$ that is both sparse and incompressible by $\leq_m^P$-reductions. Fix such a language A. By Lemma 5.16, $\mu_p(P_m^{-1}(A)) = 0$. Also, since A is sparse, the main result of [23] implies that $\mu_p(P_m(A)) = 0$.

(b) If $A \in \mathrm{P} - \{\emptyset, \{0,1\}^*\}$, then $\mu(P_m(A) \mid \mathrm{E}) = \mu_p(P_m(A)) = 0$, but $\mu_p(P_m^{-1}(A)) \neq 0$ and $\mu(P_m^{-1}(A) \mid \mathrm{E}) \neq 0$.

(c) If A is $\leq_m^P$-complete for E, then $\mu(P_m^{-1}(A) \mid \mathrm{E}) = \mu_p(P_m^{-1}(A)) = 0$ by Theorem 5.17, but $\mu(P_m(A) \mid \mathrm{E}) = \mu(\mathrm{E} \mid \mathrm{E}) \neq 0$.

Similar examples can be given for E_2.

6. Complexity cores: Upper bound. In this section we give an explicit *upper* bound on the sizes of complexity cores of languages that are $\leq_m^P$-hard for E. This bound implies that $\leq_m^P$-complete languages for E have *unusually small* complexity cores, for languages in E.

THEOREM 6.22. *For every $\leq_m^P$-hard language H for* E, *there exist $B, D \in \mathrm{DTIME}(2^{4n})$ such that D is dense and $B = H \cap D$.*

Proof. By Corollary 4.8, there is a language in E that is incompressible by $\leq_m^P$-reductions. In fact, Meyer's construction [26] shows that there is a language $A \in \mathrm{DTIME}(5^n)$ that is incompressible by $\leq_m^P$-reductions. As in Fact 4.10 and Theorem 4.12, this idea has often been used to establish *lower* bounds on the complexities of $\leq_m^P$-hard languages. Here we use it to establish an *upper* bound.

The following simple notation is useful here. The *nonreduced image* of a language $S \subseteq \{0,1\}^*$ under a function $f : \{0,1\}^* \to \{0,1\}^*$ is

$$f^{\geq}(S) = \{f(x) \mid x \in S \text{ and } |f(x)| \geq |x|\}.$$

Note that

$$f^{\geq}(f^{-1}(S)) = S \cap f^{\geq}(\{0,1\}^*)$$

for all f and S.

Let H be $\leq_m^P$-hard for E. Then there is a $\leq_m^P$-reduction f of A to H. Let $B = f^{\geq}(A)$, $D = f^{\geq}(\{0,1\}^*)$. Since $A \in \mathrm{DTIME}(5^n)$ and $f \in \mathrm{PF}$, it is clear that $B, D \in \mathrm{DTIME}(10^n) \subseteq \mathrm{DTIME}(2^{4n})$.

Fix a polynomial q and a real number $\epsilon > 0$ such that $|f(x)| \leq q(|x|)$ for all $x \in \{0,1\}^*$ and $q(n^{2\epsilon}) < n$ a.e. Let $W = \{x \mid |f(x)| < |x|\}$. Then, for all sufficiently large $n \in \mathbf{N}$,

writing $m = \lfloor n^{2\epsilon} \rfloor$, we have

$$\begin{aligned} f(\{0,1\}^{\leq m}) - \{0,1\}^{<m} &\subseteq f(\{0,1\}^{\leq m}) - f(W_{\leq m}) \\ &\subseteq f^{\geq}(\{0,1\}^{\leq m}) \\ &\subseteq D_{\leq q(m)} \\ &\subseteq D_{\leq n}, \end{aligned}$$

whence

$$\begin{aligned} |D_{\leq n}| &\geq |f(\{0,1\}^{\leq m})| - |\{0,1\}^{<m}| \\ &\geq |\{0,1\}^{\leq m}| - |C_f| - |\{0,1\}^{<m}| \\ &= 2^m - |C_f|. \end{aligned}$$

Since $|C_f| < \infty$, it follows that $|D_{\leq n}| > 2^{n^\epsilon}$ for all sufficiently large n. Thus D is dense.

Finally, note that $B = f^{\geq}(A) = f^{\geq}(f^{-1}(H)) = H \cap f^{\geq}(\{0,1\}^*) = H \cap D$. This completes the proof of Theorem 6.22. □

We now use Theorem 6.22 to prove our upper bound on the size of complexity cores for hard languages.

THEOREM 6.23. *Every DTIME(2^{4n})-complexity core of every $\leq_m^P$-hard language for* E *has a dense complement.*

Proof. Let H be $\leq_m^P$-hard for E and let K be a DTIME(2^{4n})-complexity core of H. Choose B, D for H as in Theorem 6.22. Fix machines M_B and M_D that decide B and D, respectively, with $\text{time}_{M_B}(x) = O(2^{4|x|})$ and $\text{time}_{M_D}(x) = O(2^{4|x|})$. Let M be a machine that implements the following algorithm.

begin
  input x;
  if $M_D(x)$ accepts
    then simulate $M_B(x)$
    else run forever
end M.

Then $x \in D \Rightarrow M(x) = [\![x \in B]\!] = [\![x \in H \cap D]\!] = [\![x \in H]\!]$ and $x \notin D \Rightarrow M(x) = \perp \leq [\![x \in H]\!]$, so M is consistent with H. Also, there is a constant $c \in \mathbf{N}$ such that for all $x \in D$,

$$\text{time}_M(x) \leq c \cdot 2^{4n} + c.$$

Since K is a DTIME(2^{4n})-complexity core of H, it follows that $K \cap D$ is finite. But D is dense, so this implies that $D - K$ is dense, whence K^c is dense. □

Note that Theorem 5.17 follows from Corollary 4.9 and Theorem 6.23, but that Theorem 6.23 tells us more.

The main construction of [19] shows that for every $c \in \mathbf{N}$, there is a language H that is weakly $\leq_m^P$-hard for E and has $\{0,1\}^*$ as a DTIME(2^{cn})-complexity core. Thus, in contrast with the lower bound given by Theorem 4.12, the upper bound given by Theorem 6.23 cannot be extended to weakly $\leq_m^P$-hard languages.

Finally, we note that the upper bound given by Theorem 6.23 is tight.

THEOREM 6.24. *Let $c \in \mathbf{N}$ and $0 < \epsilon \in \mathbf{R}$.*

1. E *has a $\leq_m^P$-complete language with a DTIME(2^{cn})-complexity core K that satisfies $|K_{\leq n}| > 2^{n+1} - 2^{n^\epsilon}$ a.e.*

2. E_2 *has a $\leq_m^P$-complete language with a DTIME(2^{n^c})-complexity core K that satisfies $|K_{\leq n}| > 2^{n+1} - 2^{n^\epsilon}$ a.e.*

Proof. We prove the result for E. The proof for E_2 is similar.

Let A be a language that is $\leq_m^P$-complete for E and let $k = \lceil \frac{2}{\epsilon} \rceil$. By Corollary 4.9, fix a language $B \in E$ that has $\{0, 1\}^*$ as a DTIME(2^{cn})-complexity core. Let

$$D = \{x10^{|x|^k} | x \in \{0, 1\}^*\}$$

and define the languages

$$C = (B - D) \cup \{x10^{|x|^k} \mid x \in A\}$$

and

$$K = D^c.$$

It is clear that C is $\leq_m^P$-complete for E. Also, for all sufficiently large n,

$$|D_{\leq n}| = \sum_{m=0}^{n} |D_{=m}| \leq \sum_{m=0}^{n} 2^{m^{\frac{1}{k}}} \leq (n+1)2^{n^{\frac{1}{k}}} \leq (n+1)2^{n^{\frac{\epsilon}{2}}} < 2^{n^\epsilon} - 1,$$

so

$$|K_{\leq n}| = 2^{n+1} - 1 - |D_{\leq n}| > 2^{n+1} - 2^{n^\epsilon} \text{ a.e.}$$

We complete the proof by showing that K is a DTIME(2^{cn})-complexity core for C. For this, let $s \in \mathbf{N}$, let M be a machine that is consistent with C, and define the fast set

$$F = \{x \mid \text{time}_M(x) \leq a \cdot 2^{c|x|} + a\}.$$

It suffices to prove that $|K \cap F| < \infty$.

Let $\hat{M}$ be a machine (designed in the obvious way) such that, for all $y \in \{0, 1\}^*$,

$$\hat{M}(y) = \begin{cases} M(y) & \text{if } y \notin D, \\ \perp & \text{if } y \in D. \end{cases}$$

Then $\hat{M}$ is consistent with B (because $B - D = C - D$ and M is consistent with C) and $\{0, 1\}^*$ is a DTIME(2^{cn})-complexity core for B, so the fast set

$$\hat{F} = \{x \mid \text{time}_{\hat{M}}(x) \leq (a+1)2^{c|x|} + a\}$$

is finite. Since $K \cap F = F - D$ and $(F - D) - \hat{F}$ is finite, it follows that $|K \cap F| < \infty$, completing the proof. □

7. Conclusion. In this paper we have investigated measure-theoretic aspects of the $\leq_m^P$-reducibility structure of the exponential time complexity classes E and E_2. Among other things, we have proven the following. (For simplicity we only consider the class E.)

(i) Every weakly $\leq_m^P$-hard language for E has a dense exponential complexity core (Theorem 4.12).

(ii) For every language $A \in E$, at least one of the spans $P_m(A)$, $P_m^{-1}(A)$ has resource-bounded measure 0 (Theorem 5.15, the Small Span Theorem). Thus the $\leq_m^P$-hard languages for E form a p-measure 0 set (Theorem 5.17), *every* $\leq_m^P$-degree has measure 0 in E (Theorem 5.19), and the $\leq_m^P$-complete languages for NP form a set of measure 0 in E (Corollary 5.21).

(iii) Every DTIME(2^{4n})-complexity core of every $\leq_m^P$-hard language for E has a dense complement (Theorem 6.23). Since almost every language in E has $\{0, 1\}^*$ as a DTIME(2^{4n})-complexity core (Corollary 4.9), this says that in E the $\leq_m^P$-complete languages are *unusually simple*, in the sense that they have *unusually small* complexity cores.

It is reasonable to conjecture that most of our results hold with $\leq_m^P$ replaced by $\leq_T^P$, but investigating this may be difficult. For example, consider Theorem 5.17. Bennett and Gill [5] have shown that $P_T^{-1}(A)$ has (classical) measure 1 for all $A \in$ BPP. Thus we cannot prove that the $\leq_T^P$-hard languages for E form a measure 0 set without also proving that E $\not\subseteq$ BPP.

REFERENCES

[1] K. Ambos-Spies, *Randomness, relativizations, and polynomial reducibilities*, in Proceedings of the First Structure in Complexity Theory Conference, Springer-Verlag, Berlin, 1986, pp. 23–34.

[2] J. L. Balcázar, J. Díaz, and J. Gabarró, *Structural Complexity I*, Springer-Verlag, Berlin, 1988.

[3] ———, *Structural Complexity II*, Springer-Verlag, Berlin, 1990.

[4] J. L. Balcázar and U. Schöning, *Bi-immune sets for complexity classes*, Math. Systems Theory, 18 (1985), pp. 1–10.

[5] C. H. Bennett and J. Gill, *Relative to a random oracle A*, $P^A \neq NP^A \neq co\text{-}NP^A$ *with probability* 1, SIAM J. Comput., 10 (1981), pp. 96–113.

[6] L. Berman, *On the structure of complete sets: Almost everywhere complexity and infinitely often speedup*, in Proceedings of the Seventeenth Annual Conference on Foundations of Computer Science, IEEE Computer Society Press, 1976, pp. 76–80.

[7] L. Berman and J. Hartmanis, *On isomorphism and density of NP and other complete sets*, SIAM J. Comput., 6 (1977), pp. 305–322.

[8] R. Book and D.-Z. Du, *The existence and density of generalized complexity cores*, J. Assoc. Comput. Mach., 34 (1987), pp. 718–730.

[9] R. Book, D.-Z Du, and D. Russo, *On polynomial and generalized complexity cores*, in Proceedings of the Third Structure in Complexity Theory Conference, 1988, pp. 236–250.

[10] D.-Z. Du, *Generalized complexity cores and levelability of intractable sets*. Ph.D. thesis, University of California, Santa Barbara, CA, 1985.

[11] D.-Z. Du and R. Book, *On inefficient special cases of NP-complete problems*, Theoret. Comput. Sci., 63 (1989), pp. 239–252.

[12] S. Even, A. Selman, and Y. Yacobi, *Hard core theorems for complexity classes*, J. Assoc. Comput. Mach., 35 (1985), pp. 205–217.

[13] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. H. Freeman and Company, San Francisco, CA, 1979.

[14] D. T. Huynh, *On solving hard problems by polynomial-size circuits*, Inform. Process. Lett., 24 (1987), pp. 171–176.

[15] R. M. Karp, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[16] L. A. Levin, *Universal sequential search problems*, Problems Inform. Transmission, 9 (1973), pp. 265–266.

[17] J. H. Lutz, *Almost everywhere high nonuniform complexity*, J. Comput. System Sci., 44 (1992), pp. 220–258.

[18] ———, *The quantitative structure of exponential time*, in Proceedings of the Eighth Structure in Complexity Theory Conference, IEEE Computer Society Press, 1993, pp. 158–175.

[19] ———, *Weakly hard problems*, SIAM J. Comput., to appear. See also Proceedings of the Ninth Structure in Complexity Theory Conference, 1994, IEEE Computer Society Press, pp. 146–161.

[20] ———, *Intrinsically pseudorandom sequences*, in preparation.

[21] ———, *Resource-bounded measure*, in preparation.

[22] J. H. Lutz and E. Mayordomo, *Cook versus Karp-Levin: Separating completeness notions if NP is not small*, Theoret. Comput. Sci., to appear. See also Proceedings of the Eleventh Symposium on Theoretical Aspects of Computer Science, Springer-Verlag, New York, 1994, pp. 415–426.

[23] J. H. Lutz and E. Mayordomo, *Measure, stochasticity, and the density of hard languages*, SIAM J. Comput., 23 (1994), pp. 762–779.

[24] N. Lynch, *On reducibility to complex or sparse sets*, J. Assoc. Comput. Mach., 22 (1975), pp. 341–345.

[25] E. MAYORDOMO, *Almost every set in exponential time is P-bi-immune*, Theoret. Comput. Sci., to appear. Also in Seventeenth International Symposium on Mathematical Foundations of Computer Science, Springer-Verlag, New York, 1992, pp. 392–400.
[26] A. R. MEYER, 1977, reported in [7].
[27] P. ORPONEN, *A classification of complexity core lattices*, Theoret. Comput. Sci., 70 (1986), pp. 121–130.
[28] P. ORPONEN AND U. SCHÖNING, *The density and complexity of polynomial cores for intractable sets*, Inform. Control, 70 (1986), pp. 54–68.
[29] D. A. RUSSO AND P. ORPONEN, *On P-subset structures*, Math. Systems Theory, 20 (1987), pp. 129–136.

SIAM J. COMPUT.
Vol. 24, No. 2, pp. 296–317, April 1995

008

A GENERAL APPROXIMATION TECHNIQUE FOR CONSTRAINED FOREST PROBLEMS*

MICHEL X. GOEMANS† AND DAVID P. WILLIAMSON‡

Abstract. We present a general approximation technique for a large class of graph problems. Our technique mostly applies to problems of covering, at minimum cost, the vertices of a graph with trees, cycles, or paths satisfying certain requirements. In particular, many basic combinatorial optimization problems fit in this framework, including the shortest path, minimum-cost spanning tree, minimum-weight perfect matching, traveling salesman, and Steiner tree problems.

Our technique produces approximation algorithms that run in $O(n^2 \log n)$ time and come within a factor of 2 of optimal for most of these problems. For instance, we obtain a 2-approximation algorithm for the minimum-weight perfect matching problem under the triangle inequality. Our running time of $O(n^2 \log n)$ time compares favorably with the best strongly polynomial exact algorithms running in $O(n^3)$ time for dense graphs. A similar result is obtained for the 2-matching problem and its variants. We also derive the first approximation algorithms for many NP-complete problems, including the nonfixed point-to-point connection problem, the exact path partitioning problem, and complex location-design problems. Moreover, for the prize-collecting traveling salesman or Steiner tree problems, we obtain 2-approximation algorithms, therefore improving the previously best-known performance guarantees of 2.5 and 3, respectively [*Math. Programming*, 59 (1993), pp. 413–420].

Key words. approximation algorithms, combinatorial optimization, matching, Steiner tree problem, T-joins, traveling salesman problem

AMS subject classifications. 68Q25, 90C27

1. Introduction.

Given a graph $G = (V, E)$, a function $f : 2^V \to \{0, 1\}$, and a nonnegative cost function $c : E \to \mathbb{Q}_+$, we consider the following integer program:

$$
\begin{aligned}
&\text{Min} && \sum_{e \in E} c_e x_e && \\
&\text{subject to:} && && \\
(IP) \qquad & && x(\delta(S)) \geq f(S) && \emptyset \neq S \subset V \\
& && x_e \in \{0, 1\} && e \in E
\end{aligned}
$$

where $\delta(S)$ denotes the set of edges having exactly one endpoint in S and $x(F) = \sum_{e \in F} x_e$. The integer program (IP) can be interpreted as a very special type of covering problem in which we need to find a minimum-cost set of edges that cover all cutsets $\delta(S)$ corresponding to sets S with $f(S) = 1$. The minimal solutions to (IP) are incidence vectors of forests. We therefore refer to the graph problem associated with (IP) as *a constrained forest problem*. Let (LP) denote the linear programming relaxation of (IP) obtained by relaxing the integrality restriction on the variables x_e to $x_e \geq 0$. For the most part we will consider constrained forest problems corresponding to *proper* functions; that is, a function $f : 2^V \to \{0, 1\}$ such that the following properties hold:

(i) [Symmetry] $f(S) = f(V - S)$ for all $S \subseteq V$; and
(ii) [Disjointness] If A and B are disjoint, then $f(A) = f(B) = 0$ implies $f(A \cup B) = 0$.

*Received by the editors January 11, 1993; accepted for publication (in revised form) October 6, 1993. This research was partially supported by Defense Advanced Research Project Agency contract N00014-89-J-1988.

†Department of Mathematics, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. Additional support provided by Air Force contract AFOSR-89-0271.

‡School of Operations Research and Industrial Engineering, Cornell University, Ithaca, New York 14853. This research was conducted while the author was a graduate student at the Massachusetts Institute of Technology. Additional support was provided by a National Science Foundation graduate fellowship.

We also assume that $f(V) = 0$. Many interesting families of forests can be modelled by (IP) with proper functions. In Table 1, we have indicated some examples of proper functions along with the corresponding set of minimal forests. Thus the minimum-cost spanning tree, shortest path, Steiner tree, and T-join problems (see §3 for definitions) can be stated as *proper* constrained forest problems; that is, they can be modelled as (IP) with a proper function. Many more complex combinatorial optimization problems, such as the nonfixed point-to-point connection problem and the generalized Steiner tree problem, are also proper constrained forest problems.

TABLE 1
Examples of proper functions and proper constrained forest problems.

Input	$f(S)$	Minimal forests
	$f(S) = 1 \quad \forall S$	Spanning trees
$s, t \in V$	$f(S) = \begin{cases} 1 & \lvert S \cap \{s, t\}\rvert = 1 \\ 0 & \text{otherwise} \end{cases}$	s-t paths
$T \subseteq V$	$f(S) = \begin{cases} 1 & \emptyset \neq S \cap T \neq T \\ 0 & \text{otherwise} \end{cases}$	Steiner trees with terminals T
$T \subseteq V$	$f(S) = \begin{cases} 1 & \lvert S \cap T\rvert \text{ odd} \\ 0 & \text{otherwise} \end{cases}$	T-joins

Since many proper constrained forest problems are NP-complete, we focus our attention on heuristics. If a heuristic algorithm for an optimization problem delivers a solution guaranteed to be within a factor of α of optimal, it is said to have a *performance guarantee* of α. Furthermore, if it runs in polynomial time, it is called an α-approximation algorithm. In this paper, we present a $(2 - \frac{2}{|A|})$-approximation algorithm for proper constrained forest problems, where $A = \{v \in V : f(\{v\}) = 1\}$. Our algorithm runs in $O(\min(n^2 \log n, mn\alpha(m, n)))$ time, where $n = |V|$, $m = |E|$, and α is the inverse Ackermann function. For the sake of the analysis, we implicitly construct a feasible solution to the dual linear program to (LP), and we prove that the value of our approximate integral primal solution is within a factor of $2 - \frac{2}{|A|}$ of the value of this dual solution. Therefore, for all proper functions f, the ratio between the optimal values of (IP) and (LP) is upper bounded by $2 - \frac{2}{|A|}$. This result can be contrasted with the various logarithmic upper bounds on the ratio between the optimal values of general integer covering problems and their fractional counterparts (Johnson [19], Lovász [27], and Chvátal [5]).

Our algorithm can be characterized in several ways. It is an *adaptive greedy* algorithm in which, at every iteration, the edge with minimum *reduced* cost is selected. It is adaptive in the sense that the reduced costs are updated throughout the execution of the algorithm. It can also be seen as a primal-dual algorithm in which, alternately, primal and dual updates are performed.

Our approximation algorithm generalizes many classical exact and approximate algorithms. When applied to the spanning tree problem, it reduces to Kruskal's greedy algorithm [23]. For the s-t shortest path problem, our algorithm is reminiscent of the variant of Dijkstra's algorithm that uses bidirectional search (Nicholson [28]). The algorithm is exact in these two cases. For the Steiner tree problem, we obtain the minimum spanning tree heuristic whose many variants have been described in the literature (see [39]). In the case of the generalized Steiner tree problem, our algorithm simulates Agrawal, Klein, and Ravi's 2-approximation algorithm [1]. Their algorithm was instrumental in motivating our work. In particular, we generalize their use of duality from generalized Steiner trees to all proper constrained forest problems. In the process, we make their use of linear programming duality explicit and

provide some conceptual simplifications since neither our algorithm nor its analysis require contractions, recursive calls to construct the forest or subdivisions of edges, as is used in the presentation of Agrawal, Klein, and Ravi [1].

One important consequence of the algorithm is that it can be turned into a 2-approximation algorithm for the minimum-weight perfect matching problem given that the edge costs obey the triangle inequality. Our running time of $O(n^2 \log n)$ time is faster than the currently best-known algorithms that solve the problem exactly (due to Gabow [11] and Gabow and Tarjan [13]) on all but very sparse graphs. In addition, our algorithm improves upon all known approximation algorithms for this problem in either running time or performance guarantee.

Given the triangle inequality, the algorithm can also be turned into an approximation algorithm for related problems involving cycles or paths, instead of trees. This observation allows us to consider additional problems such as the traveling salesman problem, Hamiltonian location problems [25], and many other problems. Our algorithm can also be extended to handle some nonproper constrained forest problems. In general, our technique applies to many NP-complete problems arising in the design of communication networks, VLSI design, and vehicle routing. We have also been able to apply the technique to the prize-collecting traveling salesman problem (given the triangle inequality) and the prize-collecting Steiner tree problem, thereby deriving the first 2-approximation algorithms for these problems.

The rest of the paper is structured as follows. In §2, we describe our approximation algorithm for proper constrained forest problems. We also present its analysis and an efficient implementation. In §3, we describe how the algorithm can be applied to the various proper constrained forest problems mentioned above. In §4, we show how to extend the algorithm and proof techniques to other problems, including the prize-collecting traveling salesman problem. We discuss previous work for particular constrained forest problems in §§3 and 4. We conclude in §5 with a discussion of subsequent work.

2. The algorithm for proper constrained forest problems.

2.1. Description. The main algorithm is shown in Fig. 1. The algorithm takes as input an undirected graph $G = (V, E)$, edge costs $c_e \geq 0$ for all $e \in E$, and a proper function f. The algorithm produces as output a set of edges F' whose incidence vector of edges is feasible for (IP). The basic structure of the algorithm involves maintaining a forest F of edges, which is initially empty. The edges of F will be candidates for the set of edges to be output. The algorithm loops, in every iteration selecting an edge (i, j) between two distinct connected components of F, then merging these two components by adding (i, j) to F. The loop terminates when $f(C) = 0$ for all connected components C of F; since $f(V) = 0$, the loop will finish after at most $n - 1$ iterations. The set F' of edges that are output consists of only the edges of F needed to meet the covering requirements. More precisely, if an edge e can be removed from F such that $f(C) = 0$ for all components C of $F - e$, then e is omitted from F'.

The approximation properties of the algorithm will follow from the way we choose the edge each iteration. The decision is based on a greedy construction of an implicit solution to the dual of (LP). This dual is

$$
\begin{aligned}
& \text{Max} && \sum_{S \subset V} f(S) \cdot y_S && \\
& \text{subject to:} && && \\
(D) \qquad & && \sum_{S : e \in \delta(S)} y_S \leq c_e && e \in E, \\
& && y_S \geq 0 && \emptyset \neq S \subset V.
\end{aligned}
$$

Input: An undirected graph $G = (V, E)$, edge costs $c_e \geq 0$, and a proper function f
Output: A forest F' and a value LB

1 $F \leftarrow \emptyset$
2 *Comment: Implicitly set* $y_S \leftarrow 0$ *for all* $S \subset V$
3 $LB \leftarrow 0$
4 $\mathcal{C} \leftarrow \{\{v\} : v \in V\}$
5 For each $v \in V$
6 $d(v) \leftarrow 0$
7 While $\exists C \in \mathcal{C} : f(C) = 1$
8 Find edge $e = (i, j)$ with $i \in C_p \in \mathcal{C}$, $j \in C_q \in \mathcal{C}$, $C_p \neq C_q$ that minimizes $\epsilon = \frac{c_e - d(i) - d(j)}{f(C_p) + f(C_q)}$
9 $F \leftarrow F \cup \{e\}$
10 For all $v \in C_r \in \mathcal{C}$ do $d(v) \leftarrow d(v) + \epsilon \cdot f(C_r)$
11 *Comment: Implicitly set* $y_C \leftarrow y_C + \epsilon \cdot f(C)$ *for all* $C \in \mathcal{C}$.
12 $LB \leftarrow LB + \epsilon \sum_{C \in \mathcal{C}} f(C)$
13 $\mathcal{C} \leftarrow \mathcal{C} \cup \{C_p \cup C_q\} - \{C_p\} - \{C_q\}$
14 $F' \leftarrow \{e \in F :$ For some connected component N of $(V, F - \{e\})$, $f(N) = 1\}$

FIG. 1. *The main algorithm.*

Define an *active component* to be any component C of F for which $f(C) = 1$. In each iteration the algorithm tries to increase y_C uniformly for each active component C by a value ϵ that is as large as possible without violating the packing constraints $\sum y_S \leq c_e$. Finding such an ϵ will make a packing constraint tight for some edge (i, j) between two distinct components; the algorithm will then add (i, j) to F and merge these two components. An alternate view of this process is that the algorithm tries to find the edge (i, j) between distinct components with the minimum "reduced" cost ϵ.

We claim that the algorithm shown in Fig. 1 behaves in exactly this manner. To see that the dual solution generated in steps 2 and 11 is feasible for (D), note first that initially $\sum_{e \in \delta(S)} y_S = 0 \leq c_e$ for all $e \in E$. We show by induction that the packing constraints continue to hold. Note that it can be shown by induction that $d(i) = \sum_{S: i \in S} y_S$ for each vertex i; thus as long as vertices i and j are in different components, $\sum_{e \in \delta(S)} y_S = d(i) + d(j)$ for edge $e = (i, j)$. It follows that in a given iteration y_C can be increased by ϵ for each active component C without violating the packing constraints as long as

$$d(i) + d(j) + \epsilon \cdot f(C_p) + \epsilon \cdot f(C_q) \leq c_e,$$

for all $e = (i, j) \in E$, $i \in C_p$ and $j \in C_q$, C_p and C_q distinct. Thus the largest feasible increase in ϵ for a particular iteration is given by the formula in step 8. Once the endpoints i and j of an edge $e = (i, j)$ are in the same component, the sum $\sum_{S: e \in \delta(S)} y_S$ does not increase, so that these packing constraints will continue to hold. Hence when the algorithm terminates, the dual solution y constructed by the algorithm will be feasible for (D). By the preceding discussion, we also have that $c_e = \sum_{S: e \in \delta(S)} y_S$ for each $e \in F$. Note that the value LB computed in steps 3 and 12 corresponds to the value of the dual solution y. As the value of the dual solution is a lower bound on the optimal cost, LB provides a guarantee on the performance of the algorithm for any specific instance. Furthermore, LB will also be used in the analysis below to evaluate the worst-case performance guarantee of the algorithm.

To complete our claim that the algorithm in Fig. 1 behaves as described, we need to show that the edges removed in the final step of the algorithm are not necessary to meet the covering requirements; in other words, we need to show that F' is a feasible solution to (IP). We do this below.

Two snapshots of the algorithm for the proper function $f(S) \equiv |S| (\text{mod } 2)$ are shown in Figs. 2 and 3. The two snapshots are one iteration apart. In both figures, the cost of an edge is the Euclidean distance between its endpoints. The radius around each vertex v represents the value $d(v)$. Thick radii represent active components, thin radii inactive components. The region of the plane defined by these radii are the so-called moats of Jünger and Pulleyblank [20], [21]. The set of edges F at the end of the main loop is shown in Fig. 4, and the set of edges F' output by the algorithm is shown in Fig. 5.

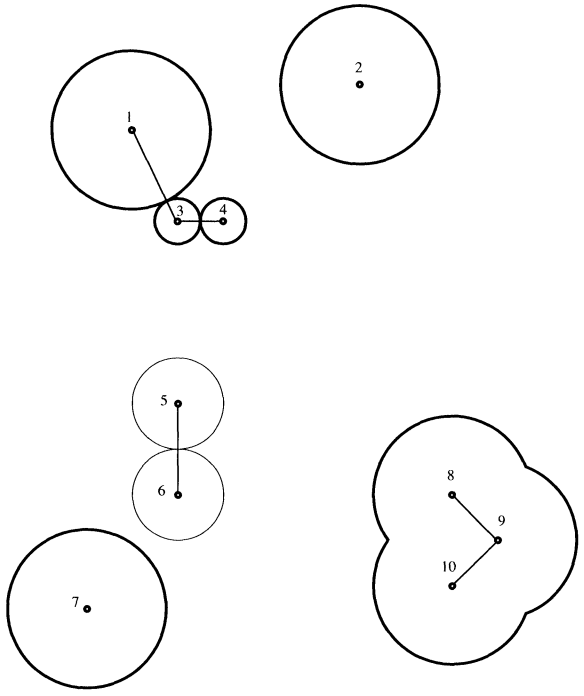


FIG. 2. *Snapshot of the algorithm.*

We can now see that the algorithm is a generalization of some classical graph algorithms. The shortest s-t path problem corresponds to the proper function $f(S) = 1$ if and only if $|S \cap \{s, t\}| = 1$. Our algorithm adds minimum-cost edges extending paths from both s and t in a manner reminiscent of Nicholson's bidirectional shortest path algorithm [28]. The main loop terminates when s and t are in the same component, and the final step of the algorithm removes all edges not on the path from s to t. Thus for this problem, whenever $y_S > 0$, $|F' \cap \delta(S)| = 1$, and whenever $e \in F'$, $\sum_{S:e \in \delta(S)} y_S = c_e$. In other words, the primal and dual feasible solutions F' and y obey the complementary slackness conditions; hence the solutions are optimal. Note that the edge removal step is necessary to obtain a good performance guarantee in this case; this statement is also true in general. The minimum-cost spanning tree problem corresponds to a proper function $f(S) = 1$ for $\emptyset \subset S \subset V$. For this function f, our algorithm reduces to Kruskal's algorithm: all components will always be active, and thus in each iteration the minimum-cost edge joining two components will be selected. Since Kruskal's algorithm produces the optimal minimum-cost spanning tree, our algorithm will also. The solutions produced do not obey the complementary slackness conditions for (LP), but induce optimal solutions for a stronger linear programming formulation of the spanning tree problem introduced by Jünger and Pulleyblank [20].

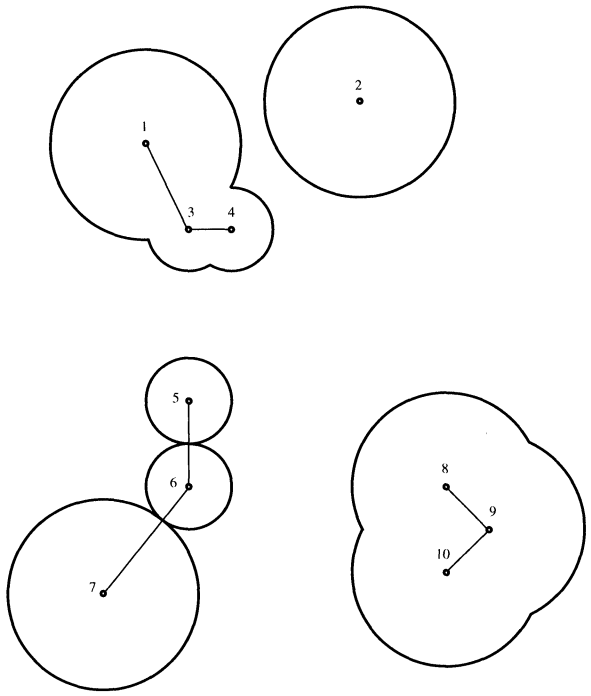


FIG. 3. *Snapshot of the algorithm one iteration later.*

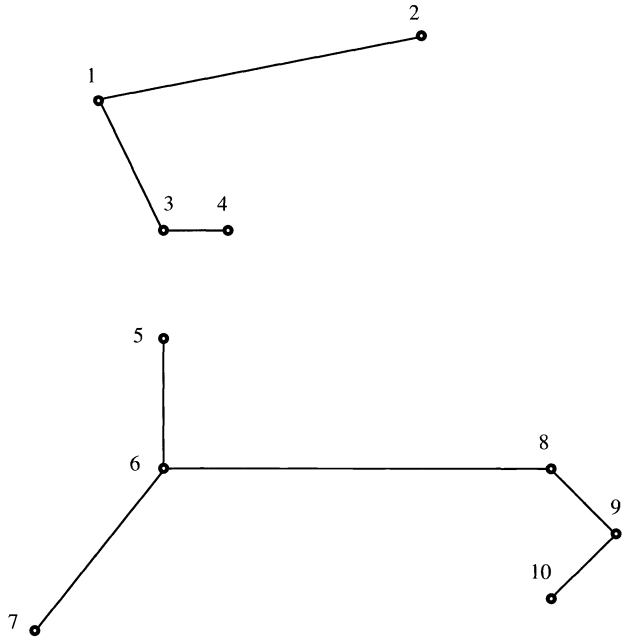


FIG. 4. *Set of edges after the main loop terminates.*

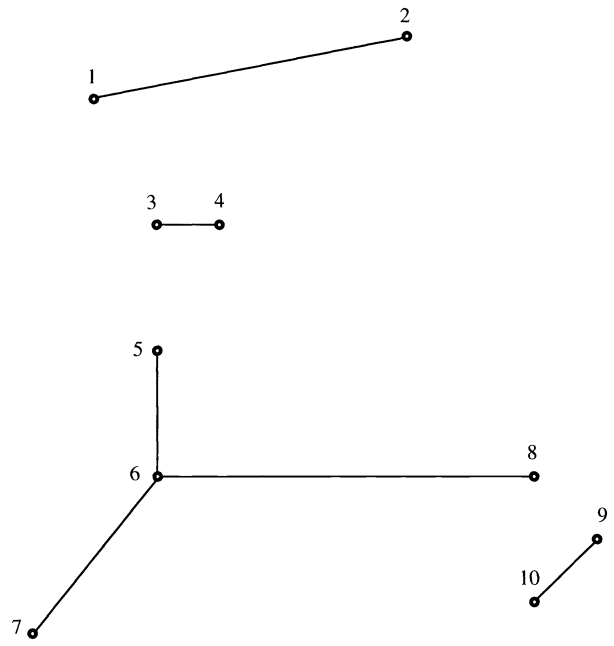


FIG. 5. *Final set of edges.*

2.2. Analysis. We now need to show that the algorithm has the properties we claim. We will begin by showing that the algorithm produces a feasible solution, and then we will turn to proving that the solution is within a factor of $(2 - \frac{2}{|A|})$ of the optimal solution. We assume throughout the ensuing discussion that F is the set of candidate edges selected by the algorithm, F' is the forest output by the algorithm, and that x' is the incidence vector of edges of F'.

OBSERVATION 2.1. *If* $f(S) = 0$ *and* $f(B) = 0$ *for some* $B \subseteq S$, *then* $f(S - B) = 0$.

Proof. By the symmetry property of f, $f(V - S) = f(S) = 0$. By disjointness, $f((V - S) \cup B) = 0$. By symmetry again, $f(S - B) = f((V - S) \cup B) = 0$. □

LEMMA 2.2. *For each connected component* N *of* F', $f(N) = 0$.

Proof. By the construction of F', $N \subseteq C$ for some component C of F. Now, let $e_1, \ldots, e_k$ be edges of F such that $e_i \in \delta(N)$ (possibly $k = 0$). Let N_i and $C - N_i$ be the two components created by removing e_i from the edges of component C, with $N \subseteq C - N_i$ (see Fig. 6). Note that since $e_i \notin F'$, it must be the case that $f(N_i) = 0$. Note also that the sets $N, N_1, N_2, \ldots, N_k$ form a partition of C. So then $f(C - N) = f(\cup_{i=1}^{k} N_i) = 0$ by disjointness. Because $f(C) = 0$, the observation above implies that $f(N) = 0$. □

THEOREM 2.3. *The incidence vector* x' *is a feasible solution to* (IP).

Proof. Suppose not, and assume that $x'(\delta(S)) = 0$ for some S such that $f(S) = 1$. Let $N_1, \ldots, N_p$ be the components of F'. In order for $x'(\delta(S)) = 0$, it must be the case that for all i, either $S \cap N_i = \emptyset$ or $S \cap N_i = N_i$. Thus $S = N_{i_1} \cup \cdots \cup N_{i_k}$ for some $i_1, \ldots, i_k$. By the lemma above, however, $f(N_i) = 0$ for all i, so $f(S) = 0$ by the disjointness of f. This contradicts our assumption that $f(S) = 1$. Therefore, x' must be a feasible solution. □

Now we will show that the algorithm has the approximation properties that we claim. For this purpose, we use the dual solution y implicitly constructed by the algorithm. Let Z_{LP}^* be the cost of the optimal solution to (LP), and let Z_{IP}^* be the cost of the optimal solution

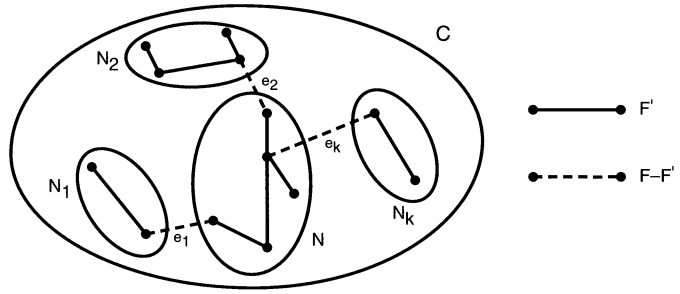


FIG. 6. *Illustration of Lemma* 2.2.

to (IP). Obviously $Z^*_{LP} \leq Z^*_{IP}$. Because y is a feasible dual solution and $y_S > 0$ only if $f(S) = 1$, it follows that $LB = \sum_{S \subset V} y_S \leq Z^*_{LP}$. We will now prove the following theorem.

THEOREM 2.4. *The algorithm in Fig.* 1 *produces a set of edges* F' *and a value* LB *such that*

$$\sum_{e \in F'} c_e \leq \left(2 - \frac{2}{|A|}\right) LB = \left(2 - \frac{2}{|A|}\right) \sum_{S \subset V} y_S \leq \left(2 - \frac{2}{|A|}\right) Z^*_{LP} \leq \left(2 - \frac{2}{|A|}\right) Z^*_{IP}.$$

Hence the algorithm is a $(2 - \frac{2}{|A|})$*-approximation algorithm for the constrained forest problem for any proper function* f.

Proof. Since we know that $\sum_{e \in F'} c_e = \sum_{e \in F'} \sum_{S : e \in \delta(S)} y_S$, by exchanging the summations we can rewrite $\sum_{e \in F'} c_e$ as $\sum_{S \subset V} y_S \cdot |F' \cap \delta(S)|$. To prove the theorem, we will show by induction on the main loop that

$$\sum_{S \subset V} y_S \cdot |F' \cap \delta(S)| \leq \left(2 - \frac{2}{|A|}\right) \sum_{S \subset V} y_S.$$

Certainly the inequality holds before the first iteration of the loop, since initially all $y_S = 0$. Consider the set $\mathcal{C}$ of components at the beginning of some iteration of the loop. The left-hand side of the inequality will increase by

$$\sum_{C \in \mathcal{C} : f(C) = 1} \epsilon \cdot |F' \cap \delta(C)|$$

in this iteration. If we can prove that this increase is bounded above by the increase of the right-hand side, namely

$$\left(2 - \frac{2}{|A|}\right) \epsilon \cdot |\mathcal{C}^1|,$$

where $\mathcal{C}^1 = \{C \in \mathcal{C} : f(C) = 1\}$, then we will be done.

The basic intuition behind the proof of this result is that the average degree of a vertex in a forest of at most $|A|$ vertices is at most $2 - \frac{2}{|A|}$. To begin, construct a graph H by considering the active and inactive components of this iteration as vertices of H, and the edges $e \in \delta(C) \cap F'$ for all $C \in \mathcal{C}$ as the edges of H. Remove all isolated vertices in H that correspond to inactive components. Note that H is a forest. We claim that no leaf in H corresponds to an inactive vertex. To see this, suppose otherwise, and let v be a leaf, C_v its associated inactive component, e the edge incident to v, and C the component of F that contains C_v. Let N and $C - N$ be the

two components formed by removing edge e from the edges of component C. Without loss of generality, say that $C_v \subseteq N$. The set $N - C_v$ is partitioned by some of the components of the current iteration; call these $C_1, \ldots, C_k$ (see Fig. 7). Since vertex v is a leaf, no edge in F' connects C_v to any C_i. Thus by the construction of F', $f(\cup C_i) = 0$. Since $f(C_v) = 0$ also, it follows that $f(N) = 0$. We know $f(C) = 0$, so by Observation 2.1 $f(C - N) = 0$ as well, and thus by the construction of F', $e \notin F'$, which is a contradiction.

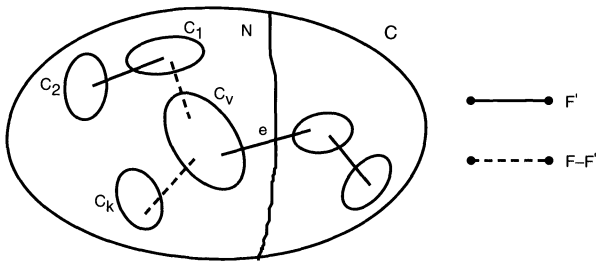


FIG. 7. *Illustration of claim that all leaves of H are active.*

In the graph H, the degree d_v of vertex v corresponding to component C must be $|\delta(C) \cap F'|$. Let N_a be the set of vertices in H corresponding to active components, so that $|N_a| = |\mathcal{C}^1|$. Let N_i be the set of vertices in H that correspond to inactive components. Then

$$\begin{aligned}\sum_{v \in N_a} d_v &= \sum_{v \in N_a \cup N_i} d_v - \sum_{v \in N_i} d_v \\ &\leq 2(|N_a| + |N_i| - 1) - 2|N_i| \\ &= 2|N_a| - 2.\end{aligned}$$

This inequality holds since H is a forest with at most $|N_a| + |N_i| - 1$ edges, and since each vertex corresponding to an inactive component has degree at least 2. Multiplying each side by ϵ, we obtain $\epsilon \sum_{v \in N_a} d_v \leq \epsilon(2|N_a| - 2)$, or

$$\epsilon \sum_{C \in \mathcal{C}^1} |F' \cap \delta(C)| \leq 2\epsilon(|\mathcal{C}^1| - 1) \leq \left(2 - \frac{2}{|A|}\right) \epsilon \cdot |\mathcal{C}^1|,$$

since the number of active components is always no more than $|A|$. Hence the theorem is proven. □

2.3. Implementing the algorithm. We now turn to the problem of implementing the algorithm efficiently and show how the algorithm can be made to run in $O(\min(n^2 \log n, mn\alpha(m, n)))$ time. We neglect the time taken to compute f from this discussion, since we can compute $f(C)$ in $O(n)$ time for all problems of interest, and since we need to perform this computation at most $O(n)$ times.

Some of the implementation details are obvious. For example, we can maintain the components C as a union-find structure of vertices. Then all merging will take at most $O(n\alpha(n, n))$ time overall, where α is the inverse Ackermann function [33]. The two main algorithmic problems arise from selecting the edge that minimizes ϵ at each iteration, and from finding the edges in F that belong in F'. We consider each of these problems separately.

As a naive approach to finding the minimum edge, we can simply use $O(m\alpha(m, n))$ time each iteration to compute the reduced cost $(c_e - d(i) - d(j))/(f(C_p) + f(C_q))$ for each edge $e = (i, j)$ and to check whether the edge spans two different components. Other loop

operations take $O(n)$ time, resulting in a running time of $O(mn\alpha(m, n))$ for the main loop, since there are at most $n - 1$ iterations.

By being somewhat more careful, we can reduce the time taken to find the minimum edge in dense graphs to $O(n \log n)$. We need three ideas for this reduced time bound. The first idea is to introduce a notion of time into the algorithm. We let the time T be 0 at the beginning of the algorithm and increment it by the value of ϵ each time through the main loop. The second idea is that instead of computing the reduced cost for an edge every time through the loop, we can maintain a priority queue of edges, where the key of an edge is the time T at which its reduced cost is expected to be zero. If we know whether the components of an edge's endpoints are active or inactive, and assume that the activity (or inactivity) will continue indefinitely, it is easy to compute this time T. Of course the activity of a component can change, but this occurs only when it is merged with another component, and only edges incident to the component are affected. In this case, we can recompute the key for each incident edge, delete the element with the old key, and reinsert it with the new key. The last idea we need for the lower time bound is that we only need to maintain a single edge between any two components. If there is more than one edge between any two components, one of the edges will always have a reduced cost no greater than that of the others; hence the others may be removed from consideration altogether.

Combining these ideas, we get the following algorithm for the main loop: first, we calculate the initial key value for each edge and insert each edge into the queue (in time $O(m \log n)$). Each time through the loop, we find the minimum edge (i, j) by extracting the minimum element from the queue. If $i \in C_p$ and $j \in C_q$, we delete all edges incident to C_p and C_q from the queue. For each component C_r different from C_p and C_q we update the keys of the two edges from C_p to C_r and C_q to C_r, select the one edge that has the minimum key value, then reinsert it into the queue. Since there are at most n components at any point in time, each iteration will have $O(n)$ queue insertions and deletions, yielding a time bound of $O(n \log n)$ per iteration, or $O(n^2 \log n)$ for the entire loop.

To compute F' from F, we iterate through the components C of F. Given a component C, we root the tree at some vertex, put each leaf of the tree in a separate list, and compute the f value for each of the leaves. An edge joining a vertex to its parent is discarded if the f value for the set of vertices in its subtree is 0. Whenever we have computed the f value for all the children of some vertex v, we concatenate the lists of all the children of v, add v to the list, and compute f of the vertices in the list. We continue this process until we have examined every edge in the tree. Since there are $O(n)$ edges, the process takes $O(n)$ time.

3. Applications of the algorithm. In this section, we list several problems to which the algorithm can be applied.

The generalized Steiner tree problem. The generalized Steiner tree problem is the problem of finding a minimum-cost forest that connects all vertices in T_i for $i = 1, \ldots, p$. The generalized Steiner tree problem is a proper constrained forest problem with $f(S) = 1$ if there exists $i \in \{1, \ldots, p\}$ with $\emptyset \neq S \cap T_i \neq T_i$ and 0 otherwise. In this case, our approximation algorithm has a performance guarantee of $2 - \frac{2}{k}$, where $k = \left|\bigcup_{i=1,\ldots,p} T_i\right|$, and simulates an algorithm of Agrawal, Klein, and Ravi [1]. Their algorithm was the first approximation algorithm for this problem.

When $p = 1$, the problem reduces to the classical Steiner tree problem. For a long time, the best approximation algorithm for this problem had a performance guarantee of $(2 - \frac{2}{k})$ (for a survey, see Winter [39]) but, very recently, Zelikovsky [40] obtained an $\frac{11}{6}$−approximation algorithm. An improved $\frac{16}{9}$-approximation algorithm based upon Zelikovsky's ideas was later proposed by Berman and Ramaiyer [3].

The performance guarantee of the algorithm can be shown to be tight for this problem. When $p = 1$, our algorithm reduces to the standard minimum-cost spanning tree heuristic (see Goemans and Bertsimas [15]). The heuristic can produce solutions which have cost $2 - \frac{2}{k}$ times the optimal cost, as is shown in [15].

The T-join problem. Given an even subset T of vertices, the T-join problem consists of finding a minimum-cost set of edges that has odd degree at vertices in T and even degree at vertices not in T. Edmonds and Johnson [9] have shown that the T-join problem can be solved in polynomial time and can be formulated by the linear program (LP) with the proper function $f(S) = 1$ if $|S \cap T|$ is odd and 0 otherwise. The edge-removing step of our algorithm guarantees that the solution produced is a T-join (see below). Using our algorithm, we obtain a $(2 - \frac{2}{|T|})$-approximation algorithm for the T-join problem.

The performance guarantee of the algorithm is tight for the T-join problem. Figure 8(a)-(c) shows an example on eight vertices in which the minimum-cost V-join has cost $4+3\epsilon$, while the solution produced by the algorithm has cost 7, yielding a worst-case ratio of approximately $\frac{7}{4} = 2 - \frac{2}{8}$. Clearly the example can be extended to larger numbers of vertices and to an arbitrary set T.

When $|T| = 2$, the T-join problem reduces to the shortest path problem. Our algorithm is exact in this case, since $2 - \frac{2}{|T|} = 1$.

The minimum-weight perfect matching problem. The minimum-weight perfect matching problem is the problem of finding a minimum-cost set of nonadjacent edges that cover all vertices. This problem can be solved in polynomial time by the original primal-dual algorithm discovered by Edmonds [7]. The fastest strongly polynomial time implementation of Edmonds' algorithm is due to Gabow [11]. Its running time is $O(n(m + n \log n))$. For integral costs bounded by C, the best weakly polynomial algorithm runs in $O(m\sqrt{n\alpha(m, n) \log n} \log nC)$ time and is due to Gabow and Tarjan [13].

These algorithms are fairly complicated and, in fact, too time-consuming for large instances that arise in practice. This motivated the search for faster approximation algorithms. Reingold and Tarjan [30] have shown that the greedy procedure has a tight performance guarantee of $\frac{4}{3}n^{0.585}$ for general nonnegative cost functions. Supowit, Plaisted, and Reingold [32] and Plaisted [29] have proposed an $O(\min(n^2 \log n, m \log^2 n))$ time approximation algorithm for instances that obey the triangle inequality. Their algorithm has a tight performance guarantee of $2 \log_3(1.5n)$. As shown by Gabow and Tarjan [13], an exact scaling algorithm for the maximum-weight matching problem can be used to obtain an $(1 + 1/n^a)$-approximation algorithm ($a \geq 0$) for the minimum-weight perfect matching problem. Moreover, if the original exact algorithm runs in $O(f(m, n) \log C)$ time, the resulting approximation algorithm runs in $O(m\sqrt{n \log n} + (1 + a) f(m, n) \log n)$. Vaidya [34] obtains a $(3 + 2\epsilon)$-approximation algorithm for minimum-weight perfect matching instances satisfying the triangle inequality. His algorithm runs in $O(n^2 \log^{2.5} n \log(1/\epsilon))$ time.

The algorithm for proper constrained forest problems can be used to approximate the minimum-weight perfect matching problem when the edge costs obey the triangle inequality. We use the algorithm with the proper function $f(S)$ being the parity of $|S|$, i.e., $f(S) = 1$ if $|S|$ is odd and 0 if $|S|$ is even. This function is the same as the one used for the V-join problem. The algorithm returns a forest whose components have even size. More precisely, the forest is a V-join, and each vertex has odd degree: if a vertex has even degree, then, by a parity argument, some edge adjacent to the vertex could have been deleted so that the resulting components have even size. Thus this edge would have been deleted in the final step of the algorithm. The forest can be transformed into a perfect matching with no increase of cost by repeatedly taking two edges (u, v) and (v, w) from a vertex v of degree three or more and replacing these edges with the edge (u, w). This procedure maintains the property

that the vertices have odd degree. After $O(n)$ iterations, each vertex has degree one. Since each iteration takes $O(1)$ time, the overall procedure gives an approximation algorithm for weighted perfect matching which runs in $O(n^2 \log n)$ time and has a performance guarantee of $2 - \frac{2}{n}$.

The performance guarantee of the algorithm is tight for this problem also, as in shown in Fig. 8(d).

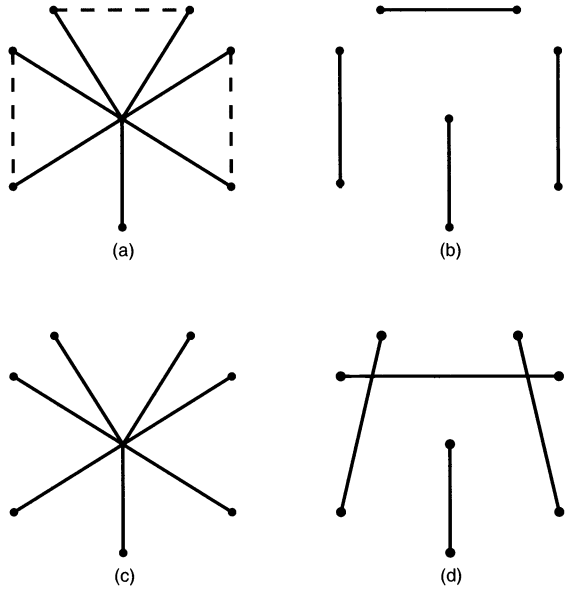


FIG. 8. *Worst-case example for V-join or matching. Graph* (a) *gives the instance: plain edges have cost* 1, *dotted edges have cost* $1 + \epsilon$, *and all other edges have cost* 2. *Graph* (b) *is the minimum-cost solution. Graph* (c) *is the set of edges found by the constrained forest algorithm, and graph* (d) *shows a bad (but possible) shortcutting of the edges to a matching.*

Point-to-point connection problems. In the point-to-point connection problem, we are given a set $C = \{c_1, \ldots, c_p\}$ of sources and a set $D = \{d_1, \ldots, d_p\}$ of destinations in a graph $G = (V, E)$ and we need to find a minimum-cost set F of edges such that each source-destination pair is connected in F [26]. This problem arises in the context of circuit switching and VLSI design. The fixed destination case in which c_i is required to be connected to d_i is a special case of the generalized Steiner tree problem where $T_i = \{c_i, d_i\}$. In the nonfixed destination case, each component of the forest F is only required to contain the same number of sources and destinations. This problem is NP-complete [26].

The nonfixed case is a proper constrained forest problem with $f(S) = 1$ if $|S \cap C| \neq |S \cap D|$ and 0 otherwise. For this problem, we obtain a $(2 - \frac{1}{p})$-approximation algorithm.

Exact partitioning problems. In the exact tree (cycle, path) partitioning problem, for a given k we must find a minimum-cost collection of vertex-disjoint trees (cycles, paths) of size k that cover all vertices. These problems and related NP-complete problems arise in the design of communication networks, vehicle routing, and cluster analysis. These problems generalize the minimum-weight perfect matching problem (in which each component must have size exactly 2), the traveling salesman problem, the Hamiltonian path problem, and the minimum-cost spanning tree problem.

We can approximate the exact tree, cycle, and path partitioning problems for instances that satisfy the triangle inequality. For this purpose, we consider the proper constrained forest

problem with the function $f(S) = 1$ if $|S| \not\equiv 0 \pmod{k}$ and 0 otherwise. Our algorithm finds a forest in which each component has a number of vertices that is a multiple of k, and such that the cost of the forest is within $2 - \frac{2}{n}$ of the optimal such forest. Obviously the cost of the optimal such forest is a lower bound on the optimal exact tree and path partitions. Given the forest, we duplicate each edge and find a tour of each component by shortcutting the resulting Eulerian graph on each component. If we remove every kth edge of the tour, starting at some edge, the tour is partitioned into paths of k nodes each. Some choice of edges to be removed (i.e., some choice of starting edge) accounts for at least $\frac{1}{k}$ of the cost of the tour, and so we remove these edges. Thus this algorithm is a $\left(4(1 - \frac{1}{k})(1 - \frac{1}{n})\right)$-approximation algorithm for the exact tree and path partitioning problems.

To produce a solution for the exact cycle partitioning problem, we add the edge joining the endpoints of each path; given the triangle inequality, this at most doubles the cost of the solution produced. We claim, however, that the algorithm is still a $\left(4(1 - \frac{1}{k})(1 - \frac{1}{n})\right)$-approximation algorithm for the cycle problem. To see that this claim is true, note that the following linear program is a linear programming relaxation of the exact cycle partitioning program, given the function f above:

$$\begin{aligned} \text{Min} \quad & \sum_{e \in E} c_e x_e & \\ \text{subject to:} & & \\ & x(\delta(S)) \geq 2f(S) & S \subset V \\ & x_e \geq 0 & e \in E. \end{aligned}$$

Its dual is

$$\begin{aligned} \text{Max} \quad & 2 \sum_{S \subset V} f(S) \cdot y_S & \\ \text{subject to:} & & \\ & \sum_{S : e \in \delta(S)} y_S \leq c_e & e \in E, \\ & y_S \geq 0 & \emptyset \neq S \subset V. \end{aligned}$$

We know the algorithm produces a solution y that is feasible for this dual such that $\sum_{e \in F'} c_e \leq (2 - \frac{2}{n}) \sum y_S$. The argument above shows how to take the set of edges F' and produce a set of edges T such that T is a solution to the exact cycle partitioning problem, and $\sum_{e \in T} c_e \leq 4(1 - \frac{1}{k}) \sum_{e \in F'} c_e$, so that

$$\sum_{e \in T} c_e \leq 8 \left(1 - \frac{1}{k}\right) \left(1 - \frac{1}{n}\right) \sum y_S.$$

Since $2 \sum y_S$ is the dual objective function, $2 \sum y_S$ is a lower bound on the cost of the optimal exact cycle partition, Z_C^*. Thus

$$\sum_{e \in T} c_e \leq 4 \left(1 - \frac{1}{k}\right) \left(1 - \frac{1}{n}\right) Z_C^*.$$

The proper functions corresponding to the nonfixed point-to-point connection problem, the T-join problem and the exact partitioning problems are all of the form $f(S) = 1$ if $\sum_{i \in S} a_i \not\equiv 0 \pmod{p}$ and 0 otherwise, for some integers a_i, $i \in V$, and some integer p.

4. Extensions. The main algorithm can be extended in a number of ways to handle nonproper functions f and even other somewhat different integer programming problems. We describe these extensions in this section.

4.1. More general functions f. We can weaken the conditions for a proper function f and obtain a modified $(2 - \frac{1}{n})$-approximation algorithm for the constrained forest problem. A number of new problems can be approximated under these weaker conditions; these problems are listed below. To solve these problems, the main algorithm must be modified to handle functions f that are not symmetric and cannot be made symmetric without violating disjointness. The central modifications that must be made involve maintaining a root vertex for each component, to cope with the asymmetry of f, and maintaining sets of vertices that must be connected in the final solution.

We omit discussion of the extended algorithm here because a recent algorithm of Williamson et al. [38] simplifies and subsumes our extended algorithm. Williamson et al. have shown how some of the results of this paper can be extended to *uncrossable* functions h. A function $h : 2^V \to \{0, 1\}$ is uncrossable if whenever $h(A) = h(B) = 1$ then either $h(A - B) = h(B - A) = 1$, or $h(A \cup B) = h(A \cap B) = 1$. Williamson et al. show that an algorithm somewhat similar to our algorithm finds solutions to (IP) for uncrossable functions that are within a factor of $2 - \frac{1}{n}$ of optimal. The algorithm can be implemented in polynomial time for many uncrossable functions, including those for which the function has the property that if $h(A) = 1$ then $h(B) = 1$ for $\emptyset \neq B \subset A$. The problems listed below fit in this category. See Williamson [36] and Goemans and Williamson [17] for a discussion of how the techniques of Williamson et al. apply to these problems.

Lower capacitated partitioning problems. The *lower capacitated partitioning problems* are like the exact partitioning problems except that each component is required to have at least k vertices rather than exactly k vertices. The lower capacitated *cycle* partitioning problem is a variant of the 2-matching problem. More precisely, the cases $k = 2$, 3, and 4 correspond to integer, binary, and triangle-free binary 2-matchings, respectively. The lower capacitated cycle partitioning problem is NP-complete for $k \geq 5$ (Papadimitriou (as reported in [6]) for $k \geq 6$ and Vornberger [35] for $k = 5$), polynomially solvable for $k = 2$ or 3 (Edmonds and Johnson [8]), while its complexity for $k = 4$ is open. Imielinska, Kalantari, and Khachiyan [18] have shown that the lower capacitated tree partitioning problem is NP-complete for $k \geq 4$.

The lower capacitated tree partitioning problem is the constrained forest problem corresponding to $f(S) = 1$ if $0 < |S| < k$ and 0 otherwise. The extended algorithm gives a $(2 - \frac{1}{n})$-approximation algorithm for this problem for any k. Furthermore, assuming the triangle inequality, this algorithm can be turned into a $(2 - \frac{1}{n})$-approximation algorithm for the lower capacitated cycle partitioning problem and a $(4 - \frac{2}{n})$-approximation algorithm for the lower capacitated path partitioning problem.

Location-design and location-routing problems. Many network design or vehicle routing problems require two levels of decisions. In the first level, the location of special vertices, such as concentrators or switches in the design of communication networks, or depots in the routing of vehicles, need to be decided. There is typically a set of possible locations and a fixed cost is associated with each of them. Once the locations of the depots are decided, the second level deals with the design or routing per se. These problems are called location-design or location-routing problems [24]. Several of these problems can be approximated using the extended algorithm. For example, we can provide a $(2 - \frac{1}{n})$-approximation algorithm for the problem in which we need to select depots among a subset D of vertices of a graph $G = (V, E)$ and cover all vertices in V with a set of cycles, each containing a selected depot [25], [24]. The goal is to minimize the sum of the fixed costs of opening our depots and the sum of the

costs of the edges of our cycles. The algorithm can also be extended to the case in which every cycle is required to have at least k vertices.

4.2. Nonnegative functions f. Using techniques from Goemans and Bertsimas [15], we can provide approximation algorithms for many functions $f : 2^V \to \mathbb{N}$, assuming that we can have multiple copies of an edge in the solution. Suppose f satisfies $f(S) = f(V - S)$ for all $S \subseteq V$, and, if A and B are disjoint, then $\max\{f(A), f(B)\} \geq f(A \cup B)$. Suppose also that f assumes at most p different nonzero values, $\rho_0 = 0 < \rho_1 < \cdots < \rho_p$. Let (IP') denote the integer program (IP) with the $x_e \in \{0, 1\}$ constraint replaced by the constraint $x_e \in \mathbb{N}$. Then we can show that there is an approximation algorithm for (IP') that comes within a factor of $2\sum_{k=1}^{p}(\rho_k - \rho_{k-1})/\rho_k$ of optimal. Note that at worst the values of f will be $0, 1, 2, 3, \ldots, f_{\max} = \max_S f(S)$, so that the performance guarantee will be at most $2\sum_{k=1}^{f_{\max}} \frac{1}{k} = O(\log f_{\max})$. The performance guarantee will also be no worse than $2p$. The algorithm for (IP') works by performing p iterations of our main algorithm. In iteration i, set $g(S) = 1$ if $f(S) \geq \rho_{p+1-i}$, $g(S) = 0$ otherwise, and call the main algorithm with function g. By the properties of f, g will be a proper function for the main algorithm. When the algorithm returns F', we make $(\rho_{p+1-i} - \rho_{p-i})$ copies of each edge, and add them to the set of edges to be output. The proof that constructing a set of edges in this way comes within a factor of $2\sum_{k=1}^{p}(\rho_k - \rho_{k-1})/\rho_k$ of optimal is essentially the same as the proof used by Goemans and Bertsimas. We can potentially reduce the number of calls to our main algorithm by using a "scaling" technique introduced by Agrawal, Klein, and Ravi [1], which requires $\lfloor \log f_{\max} \rfloor + 1$ iterations. In iteration i, we set $g(S) = 1$ if $f(S) \geq 2^{\lfloor \log f_{\max} \rfloor + 1 - i}$, $g(S) = 0$ otherwise, and call the main algorithm with the function g. We make $2^{\lfloor \log f_{\max} \rfloor + 1 - i}$ copies of the edges in the resulting F', and add them to the set of edges to be output. Using the Goemans and Bertsimas proof, it can be shown that this procedure results in a $(2\lfloor \log f_{\max} \rfloor + 2)$-approximation algorithm.

One application of allowing $f_{\max} > 1$ is the generalized Steiner network problem in which each pair of vertices i, j must be connected by r_{ij} edge-disjoint paths. In this case we want $f(S) = \max_{i \in S, j \notin S} r_{ij}$ For this particular problem, Agrawal, Klein, and Ravi [1] showed how to reduce this general case to the 0-1 case.

Williamson et al. [38] have recently shown how to approximate (IP) for functions of the type mentioned above when no edge replication is allowed.

4.3. The prize-collecting problems. The prize-collecting traveling salesman problem is a variation of the classical traveling salesman problem (TSP). In addition to the cost on the edges, we have also a penalty π_i on each vertex i. The goal is to find a tour on a subset of the vertices that minimizes the sum of the cost of the edges in the tour and the vertices not in the tour. We consider the version in which a prespecified root vertex r has to be in the tour; this is without loss of generality, since we can repeat the algorithm n times, setting each vertex to be the root. This version of the prize-collecting TSP is a special case of a more general problem introduced by Balas [2]. The prize-collecting Steiner tree problem is defined analogously. The standard Steiner tree problem can be seen to be a special case of the prize-collecting Steiner tree problem in which nonterminals have a penalty of zero, while terminals have a very large penalty (e.g., equal to the diameter of the graph).

Bienstock et al. [4] developed the first approximation algorithms for these problems. Their performance bounds are $5/2$ for the TSP version (assuming the triangle inequality) and 3 for the Steiner tree version. These approximation algorithms are not very efficient, however, since they are based upon the solution of a linear programming problem.

These problems do not fit in the framework of problems considered so far since they cannot be modelled by (IP). However, the main algorithm can be modified to give a $(2 - \frac{1}{n-1})$-

approximation algorithm for both the prize-collecting TSP (under the triangle inequality) and the prize-collecting Steiner tree problem. Moreover, these algorithms are purely combinatorial and do not require the solution of a linear programming problem as in [4]. We will focus our attention on the prize-collecting Steiner tree problem, and at the end of the section we will show how the algorithm for the tree problem can be easily modified to yield a prize-collecting TSP algorithm.

4.3.1. The prize-collecting Steiner tree. The prize-collecting Steiner tree can be formulated as the following integer program:

$$\text{Min} \quad \sum_{e \in E} c_e x_e + \sum_{T \subset V; r \notin T} z_T \left(\sum_{i \in T} \pi_i \right)$$

subject to:

$$\begin{aligned}
& x(\delta(S)) + \sum_{T \supseteq S} z_T \geq 1 && S \subset V; r \notin S \\
& \sum_{T \subset V; r \notin T} z_T \leq 1 && \\
& x_e \in \{0, 1\} && e \in E \\
& z_T \in \{0, 1\} && T \subset V; r \notin T.
\end{aligned} \tag{PC-IP}$$

Intuitively, z_T is set to 0 for all T except the set T of all vertices not spanned by the tree of selected edges. A linear programming relaxation $(PC\text{-}LP)$ of the integer program can be created by replacing the integrality constraints with the constraints $x_e \geq 0$ and $z_T \geq 0$ and dropping the constraint $\sum_T z_T \leq 1$ (in fact, including this constraint does not affect the optimal solution). The LP relaxation $(PC\text{-}LP)$ can be shown to be equivalent to the following, perhaps more natural, linear programming relaxation of the prize-collecting Steiner tree problem, which was used by the algorithm of Bienstock et al. [4]:

$$\text{Min} \quad \sum_{e \in E} c_e x_e + \sum_{i \neq r} (1 - s_i) \pi_i$$

subject to:

$$\begin{aligned}
& x(\delta(S)) \geq s_i && i \in S; r \notin S \\
& x_e \geq 0 && e \in E \\
& s_i \geq 0 && i \in V; i \neq r.
\end{aligned}$$

The dual of $(PC\text{-}LP)$ can be formulated as follows:

$$\text{Max} \quad \sum_{S: r \notin S} y_S$$

subject to:

$$\begin{aligned}
& \sum_{S: e \in \delta(S)} y_S \leq c_e && e \in E \\
& \sum_{S \subseteq T} y_S \leq \sum_{i \in T} \pi_i && T \subset V; r \notin T \\
& y_S \geq 0 && S \subset V; r \notin S.
\end{aligned} \tag{PC-D}$$

The algorithm for the prize-collecting Steiner tree problem is shown in Fig. 9. The basic structure of this algorithm is similar to that of the main algorithm. The algorithm maintains a forest F of edges, which is initially empty. Hence each vertex v is initially in its own connected

component. All components except the root r are considered active, and each vertex is initially unmarked. The algorithm loops, in each iteration doing one of two things. First, the algorithm may add an edge between two connected components of F. If the resulting component contains the root r, it becomes inactive; otherwise it is active. Second, the algorithm may decide to "deactivate" a component. Intuitively, a component is deactivated if the algorithm decides it is willing to pay the penalties for all vertices in the component. In this case, the algorithm labels each vertex in the component with the name of the component. The main loop terminates when all connected components of F are inactive. Since in each iteration the sum of the number of components and the number of active components decreases, the loop terminates after at most $2n - 1$ iterations. The final step of the algorithm removes as many edges from F as possible while maintaining two properties. First, all unmarked vertices must be connected to the root, since these vertices were never in any deactivated component and the algorithm was never willing to pay the penalty for these vertices. Second, if a vertex with label C is connected to the root, then so is every vertex with label $C' \supseteq C$.

Input: An undirected graph $G = (V, E)$, edge costs $c_{ij} \geq 0$, vertex penalties $\pi_i \geq 0$, and a root vertex r
Output: A tree F', which includes vertex r, and a set of unspanned vertices X

$F \leftarrow \emptyset$
Comment: Implicitly set $y_S \leftarrow 0$ *for all* $S \subset V$
$\mathcal{C} \leftarrow \{\{v\} : v \in V\}$
For each $v \in V$
  Unmark v
  $d(v) \leftarrow 0$
  $w(\{v\}) \leftarrow 0$
  If $v = r$ then $\lambda(\{v\}) \leftarrow 0$ else $\lambda(\{v\}) \leftarrow 1$
While $\exists C \in \mathcal{C} : \lambda(C) = 1$
  Find edge $e = (i, j)$ with $i \in C_p \in \mathcal{C}$, $j \in C_q \in \mathcal{C}$, $C_p \neq C_q$ that minimizes $\epsilon_1 = \frac{c_e - d(i) - d(j)}{\lambda(C_p) + \lambda(C_q)}$
  Find $\tilde{C} \in \mathcal{C}$ with $\lambda(\tilde{C}) = 1$ that minimizes $\epsilon_2 = \sum_{i \in \tilde{C}} \pi_i - w(\tilde{C})$
  $\epsilon = \min(\epsilon_1, \epsilon_2)$
  $w(C) \leftarrow w(C) + \epsilon \cdot \lambda(C)$ for all $C \in \mathcal{C}$
  Comment: Implicitly set $y_C \leftarrow y_C + \epsilon \cdot \lambda(C)$ *for all* $C \in \mathcal{C}$
  For all $v \in C_r \in \mathcal{C}$
    $d(v) \leftarrow d(v) + \epsilon \cdot \lambda(C_r)$
  If $\epsilon = \epsilon_2$
    $\lambda(\tilde{C}) \leftarrow 0$
    Mark all unlabelled vertices of $\tilde{C}$ with label $\tilde{C}$
  else
    $F \leftarrow F \cup \{e\}$
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{C_i \cup C_j\} - \{C_i\} - \{C_j\}$
    $w(C_p \cup C_q) \leftarrow w(C_p) + w(C_q)$
    If $r \in C_p \cup C_q$ then $\lambda(C_p \cup C_q) \leftarrow 0$ else $\lambda(C_p \cup C_q) \leftarrow 1$
F' is derived from F by removing as many edges as possible but so that the following two properties hold: (1) every unlabelled vertex is connected to r; (2) if vertex v with label C is connected to r, then so is every vertex with label $C' \supseteq C$.
X is the set of all vertices not spanned by F'.

FIG. 9. *The algorithm for the prize-collecting Steiner tree problem.*

As with the main algorithm, the choices of the algorithm are motivated by the greedy construction of an implicit solution to the dual (PC-D). Initially all dual variables are set to

zero. In each iteration of the main loop, the algorithm increases y_C for all active C by a value ϵ that is as large as possible without violating the two types of packing constraints of $(PC\text{-}D)$: $\sum_{S:e\in\delta(S)} y_S \leq c_e$ for all $e \in E$, and $\sum_{S\subseteq T} y_S \leq \sum_{i\in T} \pi_i$ for all $T \subset V$. Increasing the y_C for active C by ϵ will cause one of the packing constraints to become tight. If one of the first kind of constraints becomes tight, then it becomes tight for some edge e between two connected components of the current forest F; hence we add this edge to F. If one of the second kind of constraints becomes tight, then it becomes tight for some active component C. In this case, the algorithm chooses to deactivate C.

We claim that the algorithm shown in Fig. 9 behaves exactly in the manner described above. The claim follows straightforwardly from the algorithm's construction of y and F, and from the fact that $d(i) = \sum_{S:i\in S} y_S$ and $w(C) = \sum_{S\subseteq C} y_S$ at the beginning of each iteration. Note that the algorithm keeps track of the activity of component C by setting $\lambda(C) = 1$ if and only if component C is active.

Let Z^*_{PCLP} and Z^*_{PCIP} be the optimal solutions to $(PC\text{-}LP)$ and $(PC\text{-}IP)$ respectively. Obviously $\sum_{S\subset V} y_S \leq Z^*_{PCLP} \leq Z^*_{PCIP}$. In a manner analogous to that of Theorem 2.4, we will show the following theorem.

THEOREM 4.1. *The algorithm in Fig. 9 produces a set of edges F' and a set of vertices X whose incidence vectors are feasible for $(PC\text{-}IP)$, and such that*

$$\sum_{e\in F'} c_e + \sum_{i\in X} \pi_i \leq (2 - \frac{1}{n-1}) \sum_{S\subset V} y_S \leq (2 - \frac{1}{n-1}) Z^*_{PCIP}.$$

Hence the algorithm is a $(2 - \frac{1}{n-1})$-approximation algorithm for the prize-collecting Steiner tree problem.

Proof. It is not hard to see that the algorithm produces a feasible solution to $(PC\text{-}IP)$, since F' has no nontrivial component not containing r and the component containing r is a tree.

By the construction of F', each vertex not spanned by F' (i.e., the vertices in X) lies in some component deactivated at some point during the algorithm. Furthermore, if the vertex was in some deactivated component C, then none of the vertices of C are spanned by F'. Using these observations, plus the manner in which components are formed by the algorithm, we can partition the vertices of X into disjoint deactivated components $C_1, \ldots, C_k$. These sets are the maximal labels of the vertices in X. Since each C_j is a deactivated component, it follows that $\sum_{S\subseteq C_j} y_S = \sum_{i\in C_j} \pi_i$, and thus that the inequality to be proven is implied by $\sum_{e\in F'} c_e + \sum_j \sum_{S\subseteq C_j} y_S \leq (2 - \frac{1}{n-1}) \sum_{S\subset V} y_S$. In addition, since $c_e = \sum_{S:e\in\delta(S)} y_S$ for each $e \in F'$ by construction of the algorithm, all we need to prove is that

$$\sum_{e\in F'} \sum_{S:e\in\delta(S)} y_S + \sum_j \sum_{S\subseteq C_j} y_S \leq (2 - \frac{1}{n-1}) \sum_{S\subset V} y_S,$$

or, rewriting terms as in Theorem 2.4,

$$\sum_S y_S |F' \cap \delta(S)| + \sum_j \sum_{S\subseteq C_j} y_S \leq (2 - \frac{1}{n-1}) \sum_{S\subset V} y_S.$$

As in Theorem 2.4, this theorem can be proven by induction on the main loop. Pick any particular iteration, and let $\mathcal{C}$ be the set of active components of the iteration. Let H be the graph formed by considering active and inactive components as vertices and the edges $e \in \delta(C) \cap F'$ for active C as the edges of H. Discard all isolated inactive vertices. Let N_a denote the set of active vertices in H, N_i the set of inactive vertices, N_d the set of active

vertices corresponding to active sets contained in some C_j, and d_v the degree of a vertex v in H. Note that $N_d = \{v \in N_a : d_v = 0\}$. In this iteration, the increase in the left-hand side of the inequality is $\epsilon(\sum_{v \in N_a} d_v + |N_d|)$ while the increase in the right-hand side of the inequality is $\epsilon(2 - \frac{1}{n-1})|N_a|$. Thus we would like to prove that $(\sum_{v \in N_a} d_v + |N_d|) \leq (2 - \frac{1}{n-1})|N_a|$. Note that the degree of any vertex corresponding to an active set in some C_j is zero. Hence if we can show that $\sum_{v \in N_a - N_d} d_v \leq (2 - \frac{1}{n-1})|N_a - N_d|$, then the proof will be complete.

To do this, we show that all but one of the leaves of H must be active vertices. Suppose that v is an inactive leaf of H, adjacent to edge e, and let C_v be the inactive component corresponding to v. Further suppose that C_v does not contain the root r. Since C_v is inactive and does not contain r, it must have been deactivated. Because C_v is deactivated, no vertex in C_v is unlabelled; furthermore, since v is a leaf, no vertex in C_v can lie on the path between the root and a vertex that must be connected to the root. By the construction of F', then, $e \notin F'$, which is a contradiction. Therefore, there can be at most one inactive leaf, which must correspond to the component containing r.

Then

$$\begin{aligned}
\sum_{v \in N_a - N_d} d_v &\leq \sum_{v \in (N_a - N_d) \cup N_i} d_v - \sum_{v \in N_i} d_v \\
&\leq 2(|(N_a - N_d) \cup N_i| - 1) - (2|N_i| - 1) \\
&= 2|N_a - N_d| - 1 \\
&\leq (2 - \frac{1}{n-1})|N_a - N_d|.
\end{aligned}$$

The inequality holds since all but one of the spanned inactive vertices has degree at least two, and since the number of active components is always at most $n - 1$. □

The algorithm can be implemented in $O(n^2 \log n)$ time, by using the same techniques as were given for the main algorithm. In this case, we must also keep track of the time at which we expect each component to deactivate, and put this time into the priority queue. The only other difference from the main algorithm is the final step in which edges are deleted. This step can be implemented in $O(n^2)$ time: first we perform a depth-first search from every unmarked vertex to the root, and "lock" all the edges and vertices on this path. We then look at all the deactivated components corresponding to the labels of "locked" vertices. If one of these contains an unlocked vertex, we perform a depth-first search from the vertex to the root and lock all the edges and vertices on the path. We continue this process until each locked vertex is in deactivated components that only contain locked vertices. We then eliminate all unlocked edges. This procedure requires at most n $O(n)$ time depth-first searches.

4.3.2. The prize-collecting traveling salesman problem. To solve the prize-collecting TSP given that edge costs obey the triangle inequality, we use the algorithm shown in Fig. 10. Note that the algorithm uses the above algorithm for the prize-collecting Steiner tree problem with penalties $\pi'_i = \pi_i/2$. To see that the algorithm is a $(2 - \frac{1}{n-1})$-approximation algorithm, we need to consider the following linear programming relaxation of the problem:

$$\text{Min} \quad \sum_{e \in E} c_e x_e + \sum_{T \subset V; r \notin T} z_T (\sum_{i \in T} \pi_i)$$

subject to:

$$\begin{aligned}
& x(\delta(S)) + 2 \sum_{T \supseteq S} z_T \geq 2 && r \notin S \\
& x_e \geq 0 && e \in E \\
& z_T \geq 0 && T \subset V; r \notin T.
\end{aligned}$$

This linear program is a relaxation of an integer program similar to $(PC\text{-}IP)$ in which $z_T = 1$ for the set of vertices T not visited by the tour, and $z_T = 0$ otherwise. We relax the constraint that each vertex in the tour be visited twice to the constraint that each vertex be visited at least twice. The dual of the linear programming relaxation is

$$\text{Max} \quad 2 \sum_{S: r \notin S} y_S$$

subject to:

$$\sum_{S: e \in \delta(S)} y_S \leq c_e \qquad e \in E$$

$$2 \sum_{S \subseteq T} y_S \leq \sum_{i \in T} \pi_i \qquad T \subset V, r \notin T$$

$$y_S \geq 0 \qquad S \subset V, r \notin S.$$

Note that this dual is very similar to $(PC\text{-}D)$. The dual solution generated by the algorithm for the prize-collecting Steiner tree for penalties π' will be feasible for the dual program above with penalties π. By duality, $2\sum_{S \subset V} y_S \leq Z^*_{PCTSP}$, where Z^*_{PCTSP} is the cost of the optimal solution to the prize-collecting TSP. Given a solution F' and X to the prize-collecting Steiner tree problem, the cost of our solution to the prize-collecting TSP is at most $2\sum_{e \in F'} c_e + \sum_{i \in X} \pi_i = 2(\sum_{e \in F'} c_e + \sum_{i \in X} \pi'_i)$. Theorem 4.1 shows that $\sum_{e \in F'} c_e + \sum_{i \in X} \pi'_i \leq (2 - \frac{1}{n-1}) \sum_{S \subset V} y_S$, so that

$$2(\sum_{e \in F'} c_e + \sum_{i \in X} \pi'_i) \leq 2(2 - \frac{1}{n-1}) \sum_{S \subset V} y_S \leq (2 - \frac{1}{n-1}) Z^*_{PCTSP}.$$

Thus the cost of the solution found by the algorithm is within $(2 - \frac{1}{n-1})$ of optimal.

Input: An undirected graph $G = (V, E)$, edge costs $c_{ij} \geq 0$, vertex penalties $\pi_i \geq 0$, and a root vertex r
Output: A tour T', which includes vertex r, and a set of unspanned vertices X

1 Apply the prize-collecting Steiner tree algorithm to the problem instance with graph G, edge costs c, root r, and penalties $\pi'_i = \pi_i/2$.
2 Duplicate the edges F' of the Steiner tree returned to form an Eulerian graph T.
3 Shortcut T to form a tour T'. Let X be all vertices not in the tour.

FIG. 10. *The algorithm for the prize-collecting traveling salesman problem.*

5. Concluding remarks. The approximation techniques described in the previous sections have been applied to a number of related problems since the appearance of a preliminary version of this paper [16]. Saran, Vazirani, and Young [31] showed how to use our techniques to derive an approximation algorithm for the minimum-cost 2-edge-connected graph problem. Their algorithm has a performance guarantee of 3, equal to the performance guarantee of an earlier algorithm of Frederickson and Ja'Ja' [10] for the same problem. Klein and Ravi [22] demonstrated a 3-approximation algorithm for solving (IP) for proper functions $f : 2^V \to \{0, 2\}$. Building on some ideas of Ravi and Klein, Williamson et al. [38] devised an approximation algorithm to solve (IP) for general proper functions $f : 2^V \to \mathbb{N}$ in which the disjointness condition is replaced by the more general condition $f(A \cup B) \leq \max(f(A), f(B))$ for disjoint A, B. The performance guarantee of the algorithm is at most $2k$, where $k = \max_S f(S)$, but can be lower depending on the values taken on by f. The algorithm depends on showing that the techniques of this paper can be extended to approximate uncrossable functions, as defined in §4. Goemans et al. [14] have shown how to improve the performance guarantee of this algorithm to $2(1 + \frac{1}{2} + \cdots + \frac{1}{k})$.

Gabow, Goemans, and Williamson [12] have shown how to efficiently implement the algorithm of Williamson et al. A consequence of the implementation of Gabow, Goemans, and Williamson is an $O(n^2 + n\sqrt{m \log \log n})$ implementation for our main algorithm. Finally, we have implemented the 2-approximation algorithm for Euclidean matching problems [37]. The performance of the algorithm in this case seems to be much better than the theoretical bounds given here: on 1,400 random and structured instances of up to 131,072 vertices, the algorithm was never more than 4% away from optimal.

Acknowledgments. The authors would like to thank David Shmoys for extensive comments on a draft of this paper.

REFERENCES

[1] A. AGRAWAL, P. KLEIN, AND R. RAVI, *When trees collide: An approximation algorithm for the generalized Steiner problem on networks*, in Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, 1991, pp. 134–144; SIAM J. Comput., to appear.

[2] E. BALAS, *The prize collecting traveling salesman problem*, Networks, 19 (1989), pp. 621–636.

[3] P. BERMAN AND V. RAMAIYER, *Improved approximations for the Steiner tree problem*, in Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms, 1992, pp. 325–334.

[4] D. BIENSTOCK, M. X. GOEMANS, D. SIMCHI-LEVI, AND D. WILLIAMSON, *A note on the prize collecting traveling salesman problem*, Math. Programming, 59 (1993), pp. 413–420.

[5] V. CHVATAL, *A greedy heuristic for the set-covering problem*, Math. Oper. Res., 4 (1979), pp. 233–235.

[6] G. CORNUÉJOLS AND W. PULLEYBLANK, *A matching problem with side constraints*, Discrete Math., 29 (1980), pp. 135–159. Papadimitriou's result appears in this paper.

[7] J. EDMONDS, *Maximum matching and a polyhedron with 0,1-vertices*, J. Res. Nat. Bur. Standards B, 69B (1965), pp. 125–130.

[8] J. EDMONDS AND E. JOHNSON, *Matching: A well-solved class of integer linear programs*, in Proceedings of the Calgary International Conference on Combinatorial Structures and Their Applications, R. Guy et al., eds., Gordon and Breach, 1970, pp. 82–92.

[9] ———, *Matching, Euler tours and the Chinese postman*, Math. Programming, 5 (1973), pp. 88–124.

[10] G. N. FREDERICKSON AND J. JA'JA', *Approximation algorithms for several graph augmentation problems*, SIAM J. Comput., 10 (1981), pp. 270–283.

[11] H. N. GABOW, *Data structures for weighted matching and nearest common ancestors with linking*, in Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms, 1990, pp. 434–443.

[12] H. N. GABOW, M. X. GOEMANS, AND D. P. WILLIAMSON, *An efficient approximation algorithm for the survivable network design problem*, in Proceedings of the Third MPS Conference on Integer Programming and Combinatorial Optimization, 1993, pp. 57–74.

[13] H. N. GABOW AND R. E. TARJAN, *Faster scaling algorithms for general graph-matching problems*, J. Assoc. Comput. Mach., 38 (1991), pp. 815–853.

[14] M. GOEMANS, A. GOLDBERG, S. PLOTKIN, D. SHMOYS, E. TARDOS, AND D. WILLIAMSON, *Improved approximation algorithms for network design problems*, in Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, 1994, pp. 223–232.

[15] M. X. GOEMANS AND D. J. BERTSIMAS, *Survivable networks, linear programming relaxations and the parsimonious property*, Math. Programming, 60 (1993), pp. 145–166.

[16] M. X. GOEMANS AND D. P. WILLIAMSON, *A general approximation technique for constrained forest problems*, in Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms, 1992, pp. 307–316.

[17] M. X. GOEMANS AND D. P. WILLIAMSON, *Approximating minimum-cost graph problems with spanning tree edges*, Oper. Res. Lett., 16(1994), pp. 183–189.

[18] C. IMIELIŃSKA, B. KALANTARI, AND L. KHACHIYAN, *A greedy heuristic for a minimum-weight forest problem*, Oper. Res. Lett., 14 (1993), pp. 65–71.

[19] D. S. JOHNSON, *Approximation algorithms for combinatorial problems*, J. Comput. System Sci., 9 (1974), pp. 256–278.

[20] M. JÜNGER AND W. PULLEYBLANK, *New Primal and Dual Matching Heuristics*, Research Report 91.105, Universität zu Köln, 1991.

[21] ———, *Geometric Duality and Combinatorial Optimization*,in Jahrbuch Überblicke Mathematik 1993, S. Chatterji et al., eds., Vieweg, Braunschweig, Germany, 1993, pp. 1–24; Research Report RC 17863 (#78620), IBM Watson Research Center, 1992.

[22] P. KLEIN AND R. RAVI, *When cycles collapse: A general approximation technique for constrained two-connectivity problems*, in Proceedings of the Third MPS Conference on Integer Programming and Combinatorial Optimization, 1993, pp. 39–55. Also appears as Brown University Technical Report CS-92-30.

[23] J. KRUSKAL, *On the shortest spanning subtree of a graph and the traveling salesman problem*, Proc. Amer. Math. Soc., 7 (1956), pp. 48–50.

[24] G. LAPORTE, *Location-routing problems*, in Vehicle routing: Methods and studies, B. L. Golden and A. A. Assad, eds., North-Holland, Amsterdam, 1988, pp. 163–197.

[25] G. LAPORTE, Y. NOBERT, AND P. PELLETIER, *Hamiltonian location problems*, European J. Oper. Res., 12 (1983), pp. 82–89.

[26] C.-L. LI, S. T. MCCORMICK, AND D. SIMCHI-LEVI, *The point-to-point delivery and connection problems: Complexity and algorithms*, Discrete Appl. Math., 36 (1992), pp. 267–292.

[27] L. LOVÁSZ, *On the ratio of optimal integral and fractional covers*, Discrete Math., 13 (1975), pp. 383–390.

[28] T. NICHOLSON, *Finding the shortest route between two points in a network*, Comput. J., 9 (1966), pp. 275–280.

[29] D. A. PLAISTED, *Heuristic matching for graphs satisfying the triangle inequality*, J. Algorithms, 5 (1984), pp. 163–179.

[30] E. M. REINGOLD AND R. E. TARJAN, *On a greedy heuristic for complete matching*, SIAM J. Comput., 10 (1981), pp. 676–681.

[31] H. SARAN, V. VAZIRANI, AND N. YOUNG, *A primal-dual approach to approximation algorithms for network Steiner problems*, in Proceedings of Indo-US workshop on Cooperative Research in Computer Science, 1992, pp. 166–168.

[32] K. J. SUPOWIT, D. A. PLAISTED, AND E. M. REINGOLD, *Heuristics for weighted perfect matching*, in Proceedings of the 12th Annual ACM Symposium on Theory of Computing, 1980, pp. 398–419.

[33] R. E. TARJAN, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach., 22 (1975), pp. 215–225.

[34] P. VAIDYA. Personal communication, 1991.

[35] O. VORNBERGER, *Complexity of path problems in graphs*, Ph.D. thesis, Universität-GH-Paderborn, 1979.

[36] D. P. WILLIAMSON, *On the Design of Approximation Algorithms for a Class of Graph Problems*, Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1993. Also appears as Tech Report MIT/LCS/TR-584.

[37] D. P. WILLIAMSON AND M. X. GOEMANS, *Computational experience with an approximation algorithm on large-scale Euclidean matching instances*, in Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, 1994, pp. 355–364.

[38] D. P. WILLIAMSON, M. X. GOEMANS, M. MIHAIL, AND V. V. VAZIRANI, *A primal-dual approximation algorithm for generalized Steiner network problems*, in Proceedings of the 25th Annual ACM Symposium on Theory of Computing, 1993, pp. 708–717. To appear in Combinatorica.

[39] P. WINTER, *Steiner problem in networks: A survey*, Networks, 17 (1987), pp. 129–167.

[40] A. ZELIKOVSKY, *An 11/6-approximation algorithm for the network Steiner problem*, Algorithmica, 9 (1993), pp. 463–470.

SIAM J. COMPUT.
Vol. 24, No. 2, pp. 318–339, April 1995

009

D^{over}: AN OPTIMAL ON-LINE SCHEDULING ALGORITHM FOR OVERLOADED UNIPROCESSOR REAL-TIME SYSTEMS*

GILAD KOREN† AND DENNIS SHASHA‡

Abstract. Consider a real-time system in which every task has a value that it obtains only if it completes by its deadline. The problem is to design an on-line scheduling algorithm (i.e., the scheduler has no knowledge of a task until it is released) that maximizes the guaranteed value obtained by the system.

When such a system is underloaded (i.e., there exists a schedule for which all tasks meet their deadlines), Dertouzos [Proceedings IFIF Congress, 1974, pp. 807–813] showed that the earliest deadline first algorithm will achieve 100% of the possible value. Locke [Ph.D. thesis, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, PA] showed that earliest deadline first performs very badly, however, when the system is overloaded, and he proposed heuristics to deal with overload.

This paper presents an optimal on-line scheduling algorithm for overloaded uniprocessor systems. It is optimal in the sense that it gives the best competitive ratio possible relative to an off-line scheduler.

Key words. competitive, worst-case guarantee, deadline

AMS subject classifications. 68M20, 68N25, 68Q20, 68Q25, 93C83

1. Introduction.

In real-time computing systems, correctness may depend on the completion time of tasks as much as their input/output behavior. Tasks in real-time systems have deadlines. If the deadline for a task is met, then the task is said to *succeed*. Otherwise it is said to have *failed*.

A system is *underloaded* if there exists a schedule that will meet the deadline of every task, and *overloaded* otherwise. Scheduling underloaded systems is a well-studied topic, and several on-line algorithms have been proposed for the optimal scheduling of these systems on a uniprocessor [5], [15]. Examples of such algorithms include *earliest-deadline-first* (D) and *smallest-slack-time*[1] (SL). However, none of these classical algorithms make performance guarantees during times when the system is overloaded. In fact, Locke has experimentally demonstrated that these algorithms perform quite poorly when the system is overloaded [14].

Practical systems are prone to intermittent overloading caused by a cascading of exceptional situations. A good on-line scheduling algorithm, therefore, should give a performance guarantee in overloaded as well as underloaded circumstances.

Researchers and designers of real-time systems have devised on-line heuristics to handle overloaded situations [2], [17]. Locke proposed several clever heuristics as part of the Carnegie-Mellon University Archons project [14]. Unfortunately, those heuristics offer no performance guarantee. This paper proposes an algorithm with strong performance guarantees for a large subset of the parameters considered by Locke's algorithm.

1.1. Background.

Real-time systems may be categorized by how they react when a task fails to meet its deadline. In a *hard real-time system*, a task failure is considered intolerable. The underlying assumption is that a task failure would result in a disaster, e.g., a fly-by-wire aircraft may crash if the altimeter is read a few milliseconds too late.

*Received by the editors March 30, 1992; accepted for publication (in revised form) October 12, 1993. Supported by U.S. Office of Naval Research grants N00014-91-J-1472, N00014-92-J-1719, N00014-91-J-1472, and U.S. National Science Foundation grants IRI-89-01699 and CCR-9103953. Much of this work was done while the authors were at INRIA, Rocquencourt, France.

†Courant Institute, New York University, New York, NY 10012. Current address, Bar-Ilan University, Ramat-Gan 52900,Israel (koren@bimacs.cs.biu.ac.il).

‡Courant Institute, New York University, New York, NY 10012 (shasha@cs.nyu.edu).

[1]The *slack time* of a task is the distance to its deadline minus its remaining computation time. Hence, the slack time of a task is a measure of its urgency—a task with small slack time would have to be scheduled soon in order to meet its deadline.

A less stringent class of systems is denoted as *soft real-time systems*. In such systems, each task has a positive value. The goal of the system is to obtain as much value as possible. If a task succeeds, then the system acquires its value. If a task fails, then the system gains less value from the task [15]. In a special case of soft real-time systems, called a *firm* real-time system [7], there is no value for a task that has missed its deadline, but there is no catastrophe either. (Other papers [1] denote such deadlines as *hard*. The reader should therefore be aware of the definitional variations.)

An *on-line* scheduling algorithm is one that is given no information about a task before its release time. Different tasks models can differ in the kind of information (and its accuracy) given upon release. We assume the following: when a task is released, its value and deadline are known precisely, its computation time may be known either precisely, or, more generally, within some range. Also, preemption is allowed and task switching takes no time.

The *value density* of a task is its value divided by its computation time. The *importance ratio* of a collection of tasks is the ratio of the largest value density to the smallest value density. When the importance ratio is 1, the collection is said to have *uniform value density*, i.e., a task's value equals its computation time. We will denote the importance ratio of a collection by k.

As in [4], [8], and [18] we quantify the performance guarantee of an on-line scheduler by comparing it with a *clairvoyant* [15, p. 39] scheduling algorithm (also called an off-line scheduler). A clairvoyant scheduler has complete a priori knowledge of all the parameters of all the tasks. A clairvoyant scheduler can choose a "scheduling sequence" that will obtain the maximum possible value achievable by any scheduler.[2]

As in [4], [8], and [18] we say that an on-line algorithm has a *competitive factor* r, $0 < r \leq 1$, *if and only if* it is guaranteed to achieve a cumulative value of at least r times the cumulative value achievable by a clairvoyant algorithm on *any* set of tasks. For convenience of notation, we use *competitive multiplier* as the figure of merit. The *competitive multiplier* is defined to be "one over the competitive factor." The smaller the competitive multiplier is, the better the guarantee. Our goal is to devise on-line algorithms whose guarantee is the best possible one.

Baruah et al. [3], [4] has demonstrated, by using an adversary argument, that in the uniform value density setting there can be *no* on-line scheduling algorithm with a competitive multiplier smaller than 4.

Koren and Shasha describe in a technical report [11] an algorithm called *DD-star* (DD*) that has a competitive multiplier of 4 in the *uniform* value density case and offers 100% of the possible value in the underloaded case. This showed that the bound of 4 is tight in the uniform value density case. Wang and Mao [20] independently report a similar guarantee.

On the complexity side, Baruah et. al [3], [4] showed, for environments with an importance ratio k, a lower bound of $(1 + \sqrt{k})^2$ on the best possible competitive multiplier of an on-line scheduler. This result and some pragmatic considerations reveal the following limitations of the competitive scheduling algorithms described above:

1. The algorithms all assume a uniform value density, yet some short tasks may be more important than some longer tasks.
2. The algorithms all assume that there is no value in finishing a task after its deadline. But a slightly late task may be useful in many applications.
3. The algorithms all assume that the computation time is known upon release. However, a task program that is not straight-line may take different times during different executions.

D^{over}, the on-line scheduling algorithm presented in this paper, addresses all these limitations.

[2]Finding the maximum achievable value for such a scheduler, even in the uniprocessor case, is reducible from the knapsack problem [6]; hence is NP-hard.

2. The main results. In this paper we present an on-line scheduling algorithm called D^{over} that has an optimal competitive multiplier of $(1+\sqrt{k})^2$ for environments with importance ratio k. Hence, we show that the bound in [3] and [4] is tight for all k. D^{over} also gives 100% of the value obtainable by a clairvoyant scheduler when the system is underloaded.

D^{over} can be implemented using balanced search trees, and runs at an *amortized* cost of $O(\log n)$ time per task, where n is the maximum over all time instants of the number of tasks in the system.

We also investigated two important extensions to the task model studied earlier.

- Gradual Descent:
  We relax the *firm deadline* assumption. Tasks that complete after their deadline can still have a positive value though less than their initial value. As in Locke [14], the task's value is given by a *value function* which depends on its completion time.
  We show that under a variety of value functions an appropriate version of D^{over} has a competitive multiplier of $(1+\sqrt{k})^2$ for environments with importance ratio k.
- Situations in which the exact computation time of a task is not known:
  Suppose the on-line scheduling algorithm does not know the exact computation time of a task upon its release. However, for every task T, an upper bound on its possible computation time denoted by $c_{\max}$ is given and the actual computation time of T denoted by c satisfies

$$(1-\epsilon)\cdot c_{\max} \le c \le c_{\max}$$

  for some $0 \le \epsilon < 1$.
  We show that in that case D^{over} has a competitive multiplier of

$$(1+\sqrt{k})^2 + (\epsilon \cdot k)(1+\sqrt{k}) + 1.$$

  We also show that no on-line scheduler can guarantee 100% of the value obtainable by a clairvoyant algorithm for underloaded systems in this setting.

To conserve space, we omit the details of the above two extensions in this paper. Full details can be found in [9] and [12].

The rest of the paper is organized as follows: Section 3 introduces some notation and definitions used in the paper. Section 4 describes D^{over}. Section 5 shows that D^{over} has the optimal competitive multiplier as mentioned previously. Section 6 presents the complete performance guarantee of D^{over} with respect to underloaded and overloaded periods. The paper ends with a brief conclusion section and a discussion of open problems.

3. Notation. Before we describe the full algorithm, we need some notation. We are given a collection of tasks $T_1, T_2 \cdots T_n \cdots$ denoted by Γ. For each task T_i, its value is denoted by v_i, its release time is denoted by r_i, its computation time by c_i and its deadline by d_i.

DEFINITION 3.1.

- *Underloaded and Overloaded Systems.* A system is *underloaded* if there exists a schedule that will meet the deadline of every task and *overloaded* otherwise.
- *Executable Period.* The *executable period*, Δ_i, of the task T_i is defined to be the following interval: $\Delta_i = [r_i, d_i]$.
  By definition, T_i may be scheduled only during its executable period.

Suppose a collection of tasks is being scheduled by some scheduler S.

- *Completed Task.* A task (successfully) *completes* if before its deadline, the scheduler S gives it an amount of execution time that is equal to its computation time.
- *Preempted Task.* A task is *preempted* when the processor stops executing it, but then the task might be scheduled again and possibly complete at some later point.

- *A Ready Task.* A task is said to be *ready* at time t if its release time is before t, its deadline is after t and it neither completed nor was abandoned before t (the current executing task, if any, is always a ready task).

The earliest deadline first algorithm (hereafter, D) is described in Fig. 1.

At any given moment, schedule the ready task with the earliest deadline.

FIG. 1. D, *the earliest deadline first scheduling algorithm.*

We shall make the following assumption:

ASSUMPTION 3.2.

- *Task Model.* Tasks may enter the system at any time; their computation times and deadlines are known exactly at their time of arrival. (We weaken this assumption of exact knowledge in [12].) Nothing is known about a task before it appears.
  We do assume, however, that an upper bound on the possible importance ratio is known a priori and can be used by the on-line scheduler. (This bound is denoted by k.) Other researchers have shown that this assumption can be relaxed [16].
- *Task Switching Takes No Time.* A task can be preempted and another one scheduled for execution instantly.

Suppose that a collection of tasks Γ with importance ratio k is given.

- *Normalized Importance.* Without loss of generality, assume that the smallest importance of a task in Γ is 1. Hence if Γ has *importance ratio* of k, the highest possible value density of a task in Γ is k.
- *No Overloaded Periods of Infinite Duration.* We assume that overloaded periods of infinite duration will not occur. This is a realistic assumption, since overload is normally the result of a temporary emergency or failure. Indeed, in the uniprocessor case, Baruah et. al [3] showed that there is no competitive on-line algorithm when overloaded periods of infinite duration are possible.[3] Note that the number of tasks in Γ may be infinite provided that no infinitely long overload period is generated.[4]

4. D^{over}. In the following algorithm, there are three kinds of *events* (each causing an associated interrupt) considered:

- Task Completion: Successful termination of a task. This event has the highest priority.
- Task Release: Arrival of a new task. This event has low priority.
- Latest-start-time Interrupt: The indication that a task must immediately be scheduled in order to complete by its deadline; that is, the task's remaining computation time is equal to the time remaining until its deadline. This event has also low priority (the same as task release).

If several interrupts happen simultaneously they are handled according to their priorities. A task completion interrupt is handled before the task release and latest-start-time interrupt interrupts, which are handled in random order. It may happen that a task completion event suppresses a lower priority interrupt, e.g., if the task completion handler schedules a task

[3]Intuitively, the adversary can generate a sequence of tasks with ever-growing values. This will force any competitive scheduler to abandon the current task in favor of the next one, and so on. If the competitive scheduler attempts to complete a task in favor of a new larger one, then the adversary completes the larger one. In either case, the on-line schedule will result in a small value compared with an arbitrarily large value for a clairvoyant scheduler.

[4]For the definition of overloaded periods, see §6.

that has just reached its *LST* (latest-start-time) then the task scheduling will obviate the need for the latest-start-time interrupt.

At any given moment, the set of ready tasks[5] is partitioned into two disjoint sets: *privileged* tasks and *waiting* tasks. Whenever a task is preempted it becomes a *privileged* task. However, whenever some task is scheduled as the result of a latest-start-time interrupt *all* the ready tasks (whether preempted or never scheduled) become *waiting* tasks.

D^{over} maintains a special quantity called availtime. Suppose a new task is released into the system and its deadline is the earliest among all ready tasks. The value of availtime is the maximum computation time that can be taken by such a task without causing the current task or any of the privileged tasks to miss their deadlines.

D^{over} requires three data structures, called Q_privileged, Q_waiting and Qlst. Each entry in these data structures corresponds to a task in the system. Q_privileged contains exactly the privileged tasks and Q_waiting contains the *waiting* tasks. These two structures are ordered by the tasks' deadlines. In addition, the third structure, Qlst, contains all tasks (again, not including the current task) ordered by their latest start times.

These data structures support Insert, Delete, Min, and Dequeue operations.

- The Min operation for Q_privileged or Q_waiting returns the entry corresponding to the task with the earliest deadline among all tasks in Q_privileged or Q_waiting. For Qlst the Min operation returns the entry corresponding to the task with the earliest *LST* among all tasks in the queue. The Min operation does not modify the queue.
- A Dequeue operation on Q_privileged (or Q_waiting) deletes from the queue the element returned by Min; in addition it deletes this element from Qlst. Likewise a Dequeue operation on Qlst will delete the corresponding element from either Q_privileged if it is a *privileged* task or from Q_waiting if it is a *waiting* task.

An entry of Q_waiting and Qlst consists of a single task, whereas an entry of Q_privileged is a 3-tuple (T, Previous-time, Previous-avail) where T is a task that was previously preempted at time Previous-time. Previous-avail is the value of the variable availtime at time Previous-time. All of these data structures are implemented as balanced trees (e.g. 2–3 trees).

D^{over}'s code is depicted in Figs. 2–4.

The following is an intuitive description of the algorithm: As long as no overload is detected (i.e., there is no *LST* interrupt), D^{over} schedules in the same way as D. Tasks that are preempted during this phase in favor of a task with an earlier deadline become privileged tasks. The task with the earliest deadline (either a newly released task or a *waiting* task) will be scheduled provided that it does not cause overload when added to the privileged tasks. This proviso is always met in situations of underload. During overload, when a *waiting* task reaches its *LST*, it will cause a latest-start-time interrupt. This means that some task must be abandoned: either the task that reached its *LST* or some of the privileged tasks. The latest-start-time interrupt routine compares the value of that task against the *sum* of the values of all the privileged tasks. If its value is greater than $(1 + \sqrt{k})$ times that sum, then this task will execute on the processor while all the privileged tasks will lose their privileged status to become *waiting* tasks (these tasks might later be successfully rescheduled). Otherwise, the task reaching its *LST* is abandoned. A task T that was scheduled by a latest-start-time interrupt can be abandoned in favor of another task T' that reaches its *LST* but only if T' has at least $(1 + \sqrt{k})$ times more value than T. D^{over} returns to schedule according to D when some task scheduled by its latest-start-time interrupt completes.

The reader may be curious to know why D^{over} compares *values* rather then *value densities* and why the values are compared using the magic factor of $(1 + \sqrt{k})$? The lower bound proof [3], [4] shows why value density cannot be a good criterion for choosing which task to

[5]Excluding the currently executing task.

In the following code, *Now()* is a function that returns the current time. *Schedule*(T) is a function that gives the processor to task T. *Laxity*(T) is a function that returns the amount of time the task has left until its deadline less its remaining computation time. That is, *laxity*(T) = *deadline*(T) − (*now*()+ *remaining_computation_time*(T)). ϕ denotes the empty set.
This code includes lines manipulating *intervals*. The notion of an interval is needed for purpose of analysis only, so these lines are commented.

```
1   recentval      := 0     (* This will be the running value of privileged tasks. *)
2   availtime      := ∞
                            (* Availtime will be the maximum computation time that
3                           can be taken by a new task without causing the current
                            task or the privileged tasks to miss their deadlines. *)
4   Qlst           := φ     (* All ready tasks, ordered according to their latest
                            start time. *)
5   Q_privileged   := φ     (* The privileged tasks ordered by deadline order. *)
6   Q_waiting      := φ     (* All the waiting tasks ordered by their deadlines. *)
7   idle           := true  (* In the beginning the processor is idle. *)


8   loop
9    task completion :
10     if (both Q_privileged and Q_waiting are not empty) then
                    (* Both queues are not empty and contain together all the ready tasks.
11                  The ready task with the earliest deadline will be scheduled unless it is a
                    task of Q_waiting and it cannot be scheduled with all the privileged tasks.
                    The first element in each queue is probed by the Min operation. *)
12
13                  (T Q_privileged, t_prev, avail_prev) := Min(Q_privileged);
14
```

FIG. 2. *D^{over}—a competitive optimal on-line scheduling algorithm.*

abandon.[6] The factor of $(1 + \sqrt{k})$ happened to be the one that gave the desired result since it yields the correct ratio between the minimal value gained by D^{over} and the maximal value that might have been missed.

5. Analysis of D^{over}. In order to facilitate the analysis of D^{over} it is convenient to introduce the notation of intervals.

DEFINITION 5.1.

- *Intervals.* The intervals are created (opened) and closed according to the scheduling decisions of D^{over} and this process is depicted in the code of D^{over} in §4.
  When an interval is created (comments 37 and 59 of D^{over}) it is considered *open*, meaning that it may be extended, it is closed when a task completes while Q_privileged is empty (comments 33 and 48). A new interval would be opened when the next task is scheduled. Initially, there is no open interval. Hence, the first interval is opened when the processor first becomes nonidle.

[6] In that proof, going after high value density tasks (the short *teasers*) will give the on-line scheduler minuscule value compared to the clairvoyant scheduler that will schedule a low-value density task that has long computation time and hence big value.

```
        (* Next, compute the current value of availtime. This is the correct value
        because T Q_privileged is the task last inserted of those tasks currently in
15      Q_privileged; t_prev is the time when T Q_privileged was preempted; and
        the available computation time has decreased by the time elapsed since
        this element was inserted to the queue. *)
16
17      availtime := avail_prev − (now() − t_prev);
18      (* Probe the first element of Q_waiting and check which of the two tasks
        should be scheduled. *)
19      T_Q_waiting := Min(Q_waiting);
20      if d_Q_waiting < d Q_privileged and
        availtime≥ remaining_computation_time(T_Q_waiting) then
21        (* Schedule the task from Q_waiting. *)
22        Dequeue(Q_waiting);
23        availtime:= availtime − remaining_computation_time(T_Q_waiting) ;
24        availtime:= min(availtime, laxity(T_Q_waiting));
25        Schedule T_Q_waiting;
26      else
27        (* Schedule the task from Q_privileged. *)
28        Dequeue(Q_privileged);
29        recentval := recentval − value(T Q_privileged);
30        Schedule T Q_privileged;
31      endif   (*which task to schedule. *)
32    else if (Q_waiting is not empty) then
33      (* Q_privileged is empty. The current interval is closed here, t_close =
        now(). The first task in Q_waiting is scheduled *)
34
35      T_current := Dequeue(Q_waiting);
36      availtime:= laxity(T_current);
37      (* A new interval is created with t_begin = now().*)
38
39      Schedule T_current;
40    else if (Q_privileged is not empty)
41      (* Q_waiting is empty. The first task in Q_privileged is scheduled *)
42
43      (T_current, t_prev, avail_prev) := Dequeue(Q_privileged);
44      recentval := recentval − value(T_current);
45      availtime := avail_prev − (now() − t_prev);
46      Schedule T_current;
47    else
48      (* Both queues are empty. The interval is closed here, t_close = now(). *)
49
50      idle := true;
51      availtime:= ∞;
52    endif
53  end (*task completion *)
```

FIG. 3. D^{over} (*continued*).

```
54  task release : (* T_arrival is released. *)
55      if (idle ) then
56          Schedule T_arrival;
57          availtime := laxity(T_arrival);
58          idle  := false;
59          (* A new interval is created with t_begin = now().*)
60      else  (*T_current is executing  *)
61          if  d_arrival < d_current and
            availtime ≥ computation_time(T_arrival) then
62            (* No overload is detected, so the running task is preempted. *)
63            Insert T_current into Qlst;
64            Insert (T_current, now(), availtime) into Q_privileged;
65            (* The inserted task will be, by construction, the task with the earliest
              deadline in Q_privileged*)
66            availtime:= availtime − remaining_computation_time(T_arrival);
67            availtime:= min(availtime, laxity(T_arrival))
68            recentval := recentval + value(T_current);
69            Schedule T_arrival;
70          else  (* T_arrival has later deadline or availtime is not big enough.*)
71            (* T_arrival is to wait in Q_waiting  *)
72            Insert T_arrival into Qlst and Q_waiting;
73          endif
74      endif  (*idle  *)
75  end  (*release  *)

76  latest-start-time interrupt :
77      (* The processor is not idle and the current time is the latest start time of
        the first task in Qlst. *)
78
79      T_next = Dequeue(Qlst);
80      if (v_next >  (1 + √k) (v_current + recentval)) then
81          (*v_next is big enough; it is scheduled.  *)
82          Insert T_current into Qlst and Q_waiting;
83          Remove all privileged tasks from
            Q_privileged and insert them into Qlst and Q_waiting;
84          (* Q_privileged = φ  *)
85          recentval := 0;
86          availtime:= 0
87          Schedule T_next;
88      else  (*v_next is not big enough; it is abandoned.  *)
89          Abandon T_next;
90      endif
91  end (*LST  *)
92  end{loop }
```

FIG. 4. D^{over} (*continued*).

The interval consists of the time between the point it was opened and the point it was closed. We will denote by $I = [t_{begin}, t_{close}]$ an interval I that was opened at t_{begin} and closed at t_{close}.

Note: Two intervals may overlap only at their endpoints.

- *BUSY:* Suppose D^{over} schedules a collection of tasks. Let $BUSY$ denote the time where the processor is not idle during the execution of these tasks. For simplicity, the length of $BUSY$ will also be denoted by $BUSY$.
  Note that $BUSY$ equals the union of all intervals created by D^{over}.

Suppose that a collection of tasks Γ with importance ratio k is given, and D^{over} schedules this collection. When a task is scheduled, it can have zero or positive slack time. A task may be preempted and then rescheduled several times. We will be concerned mainly with the last time a task was scheduled. For the purposes of analyzing D^{over}, we will partition the collection of tasks according to the question of whether the task had completed exactly at its deadline or before its deadline or failed.

- Let F (for fail) denote the set of tasks that were abandoned.
- Let S^p (for successful with positive time before the deadline) denote the set of tasks that completed successfully and that ended some positive time before their deadlines.
- Let S^0 (for successful with 0 time before the deadline) denote the set of tasks that completed successfully but ended exactly at their deadlines.

Call a task *order-scheduled* if it was scheduled by the task completion or task release handlers. Call a task *lst-scheduled* if it was scheduled as a result of a latest-start-time interrupt. (As mentioned previously, a latest-start-time interrupt is raised on a waiting task when it reaches its latest start time (*LST*), i.e. the last time when it can start executing and still complete by its deadline).

The first task in each interval is order-scheduled. The subsequent tasks (if any) in this interval may be order-scheduled or lst-scheduled. Proposition 5.2 shows that once a task is lst-scheduled, all subsequent tasks of this interval must be lst-scheduled. During an interval, several order-scheduled tasks may complete but only one lst-scheduled task can complete. (This task will also be the last task that executes in the interval.)

PROPOSITION 5.2. *According to the scheduling of D^{over}, once a task is lst-scheduled, then all subsequent tasks, in the current interval, are lst-scheduled.*

Proof. Suppose the current task, $T_{current}$, is lst-scheduled and a task, $T_{arrival}$, is released. $T_{arrival}$ will not be scheduled by the task release handler because when the current task is lst-scheduled availtime equals zero (see statement 86 of D^{over}) hence no task can be scheduled by the task release handler (see statement 61 of D^{over}). □

Let recentval(t) denote[7] recentval at time t and achievedvalue(t) denote the value achieved during the current interval before t by D^{over}. For an interval I, achievedvalue(I) is the total value obtained during I.

We partition the value obtained during I in two different ways:

- ordervalue vs. lstvalue: ordervalue(I) is the total value obtained by order-scheduled tasks that completed during I. The value obtained by lst-scheduled tasks is denoted by lstvalue(I). (There is at most one such task in any interval I.)
- zerolaxval vs. poslaxval: zerolaxval(I) denotes the total value obtained by tasks that completed at their deadlines during I (tasks in S^0). The value obtained by tasks that completed before their deadlines is denoted by poslaxval(I).

Hence, for every interval

$$\text{achievedvalue}(I) = \text{ordervalue}(I) + \text{lstvalue}(I) = \text{zerolaxval}(I) + \text{poslaxval}(I).$$

When the index (I) is omitted we refer to the entire execution. For example, ordervalue denotes the total value obtained by order-scheduled tasks summing over all intervals.

[7] In the following only recentval is a variable explicitly manipulated by D^{over}. All the others, zerolaxval, poslaxval, ordervalue and lstvalue, are introduced here to facilitate the analysis. This is why they do not reference algorithm statements.

EXAMPLE 5.3. Before the detailed analysis, let us first study an example of D^{over}'s scheduling. Consider the overloaded collection of six tasks depicted in Table 1. For notational convenience we will denote the tasks by their deadlines, hence for example T_{20} is a task with deadline at time 20. In this example we assume uniform value density (i.e., $k = 1$). D^{over} schedules the above collection as follows: In the beginning availtime is ∞ and Q_privileged is empty. First, D^{over} schedules T_{20} to run at time 0. Availtime is set to 14 since this is T_{20}'s laxity.

TABLE 1
The tasks for Example 5.3.

Task	*Release-Time*	*Computation-Time*	*Deadline*	Δ_i
T_{20}	0	6	20	[0, 20]
T_{34}	1	26	34	[1, 34]
T_{24}	1	20	24	[1, 24]
T_{18}	2	5	18	[2, 18]
T_{17}	3	2	17	[3, 17]
T_5	4	1	5	[4, 5]

At time 1, T_{34} is released into the system. Since T_{34}'s deadline is not earlier than the current task's (T_{20}), T_{34} is inserted into Q_waiting (and Qlst with *LST* equal to 8). Also at time 1, T_{24} is released. Again, since its deadline is after 20, this task is inserted into Q_waiting and Qlst with *LST* equal to 4.

At time 2, T_{18} is released. This time the current task is preempted. T_{20} is inserted into Q_privileged and Qlst with *LST* equals 16. Availtime is decremented by the computation time of T_{18}. Its new value is 9. The value of recentval is set to the value of T_{20} (6).

T_{18} executes for one time unit until time 3, when T_{17} is released. T_{17} is scheduled since its computation time (2) is smaller then availtime (9). Availtime is decremented by the computation time of T_{17}. Its new value is 7. The value of T_{18} (5) is added to recentval , which becomes 11.

At time 4, two events occur: T_{24} reaches its *LST* and T_5 is released. These events can be handled in any order and we choose to handle the latest-start-time interrupt first. T_{24} reaches its *LST* but its value is smaller than twice ($1 + \sqrt{k} = 2$) the value of the current task plus recentval ($2 + 11$). Hence, T_{24} is abandoned. T_5 is released and its deadline is earlier than the current task's (T_{17}). T_5 is scheduled since its computation time is smaller then availtime($1 < 7$). T_5 has laxity of zero which is smaller than the current availtime minus the computation time of T_5 (6). Hence, availtime is now set to 0 and recentval becomes $11 + 2 = 13$.

For reason of space we cannot describe here the rest of the execution. (Table 2 summarizes the entire execution.) In this example $S^0 = [T_5, T_{34}]$, $S^p = [T_{17}]$ and $F = [T_{18}, T_{20}, T_{24}]$. Only three tasks complete their execution and the total value obtained by D^{over} is 29. A clairvoyant scheduler can achieve a value of 34 by scheduling T_{17}, T_{20}, and T_{34}. Also notice that the system is already overloaded at time 1, but the first time an overload is "detected" by D^{over} is at time 4 because of the *LST* interrupt.

5.1. Proof strategy. Our goal is to show that D^{over} has a competitive multiplier of $(1 + \sqrt{k})^2$ for every collection of tasks with importance ratio of k. We will start by proving some lemmas about the behavior of D^{over}. Then we will try to estimate the best possible behavior of a clairvoyant algorithm by comparison to D^{over}. Our basic strategy is to bound from below what D^{over} achieves during each interval. This will lead to a global lower bound over the entire execution. Then, we bound from above what a clairvoyant scheduler can achieve during the entire execution.

TABLE 2

D^{over}'s scheduling for Example 5.3. *A new interval is opened at time* 0. T_{24} *reaches its LSTat time* 4 *and then abandoned.* T_{34} *reaches its LSTat time* 8 *and then scheduled.* T_{20} *and* T_{18} *reach their LSTs at times* 15 *and* 16 *respectively both are abandoned. The interval is closed with the completion of* T_{34} *at time* 34.

t	re-lea-sed	pre-empted (*LST*)	com-ple-ted	sch-edu-led	availtime	Q_priv-ileged	rec-ent-val	Q_wait ing
0					∞	[]	0	[]
0	T_{20}			T_{20}	$laxity(T_{20})$	[]	0	[]
1	T_{34}				14	[]	0	$[T_{34}]$
1	T_{24}				14	[]	0	$[T_{24}, T_{34}]$
2	T_{18}	T_{20} (16)		T_{18}	$min(14-5, 13)$	$[T_{20}]$	6	$[T_{24}, T_{34}]$
3	T_{17}	T_{18} (14)		T_{17}	$min(9-2, 12)$	$[T_{18}, T_{20}]$	$5+6$	$[T_{24}, T_{34}]$
4					$min(9-2, 12)$	$[T_{18}, T_{20}]$	11	$[T_{34}]$
4	T_5	T_{17} (16)		T_5	$min(7-1, 0)$	$[T_{17}, T_{18}, T_{20}]$	$2+11$	$[T_{34}]$
5			T_5	T_{17}	$7-(5-4)=6$	$[T_{18}, T_{20}]$	$5+6$	$[T_{34}]$
6			T_{17}	T_{18}	$9-(6-3)=6$	$[T_{20}]$	6	$[T_{34}]$
8		T_{18} (15)		T_{34}	0	[]	0	$[T_{18}, T_{20}]$
15					0	[]	0	$[T_{18}]$
16					0	[]	0	[]
34			T_{34}		0	[]		[]

5.2. Some lemmas about D^{over}'s scheduling. In this section we present some technical lemmas about the behavior of D^{over}. These lemmas will be used in the next section when comparing D^{over}'s performance with that of a clairvoyant scheduler. These lemmas concern the relationship between the interval length and the value achieved by D^{over} in that interval (Lemma 5.5), as well as the relationship between the computation time and value of tasks abandoned in an interval with respect to the value achieved in the interval (Lemmas 5.6 and 5.7). Recall that BUSY is the union of all intervals (Definition 5.1).

LEMMA 5.4.

1. *For any task* T_i *in* S^0, $\Delta_i = [r_i, d_i] \subseteq BUSY$.
2. *For any task* T_i *in* F. *Suppose* T_i *was abandoned at time* t_{aban}, *then* $[r_i, t_{aban}] \subseteq BUSY$.

Proof. A processor is idle under D^{over} scheduling only if there is no *ready* task.

- A task T_i of S^0 does not complete before its deadline, hence it is a *ready* task during all its executable period. This implies that there is no idle time during the executable period of T_i.
- Similarly, a task of F is a ready task from its release time to the point at which it is abandoned. Therefore there is no idle time between its release point and its abandonment point. □

LEMMA 5.5. *For any interval* $I = [t_{begin}, t_{close}]$, *the length of* I, $t_{close} - t_{begin}$ *will satisfy*

$$t_{close} - t_{begin} \leq \mathsf{ordervalue}(I) + \left(1 + \frac{1}{\sqrt{k}}\right) \cdot \mathsf{lstvalue}(I)$$

$$= \mathsf{achievedvalue}(I) + \frac{1}{\sqrt{k}} \cdot \mathsf{lstvalue}(I).$$

Recall that ordervalue(I) *and* lstvalue(I) *are the values obtained by* D^{over} *from the order-scheduled and the lst-scheduled tasks, respectively, during* I.

Proof. An interval $I = [t_{begin}, t_{close}]$ has the following two subportions, the second of which may be empty:

1. $[t_{begin}, t_{first\_lst}]$
   From the beginning of I to the point in time, t_{first_lst}, in which the first lst-scheduled task is scheduled. During this period all tasks are order-scheduled and some may complete their execution.
   If no task is lst-scheduled in I then define t_{first_lst} to be t_{close}. In this case the second subportion is empty.
2. $[t_{first\_lst}, t_{close}]$
   During this period, all tasks are scheduled and preempted by latest-start-time interrupt. Only the last task to be scheduled completes.

If there are no lst-scheduled tasks in I then all tasks that executed from t_{begin} to t_{close} completed successfully. The value achieved is $\mathsf{ordervalue}(I)$ and is at least as big as the duration of execution.[8] Hence, the lemma is proved in this case.

Otherwise, suppose that $T_1, T_2, \cdots, T_m$ ($m \geq 1$) are the tasks that were lst-scheduled in I. Hence, T_1 was scheduled at t_{first_lst}, later it was preempted (and abandoned) by T_2 and so forth. Eventually T_m preempts T_{m-1} and completes at t_{close}, its value v_m is $\mathsf{lstvalue}(I)$.

Denote by l_i the length of the execution of T_i during the process above. T_m preempted T_{m-1} hence $v_m > (1+\sqrt{k})v_{m-1}$. Which yields[9]

$$l_{m-1} < v_{m-1} < \frac{v_m}{(1+\sqrt{k})} = \frac{\mathsf{lstvalue}(I)}{(1+\sqrt{k})}.$$

Going backward along the chain of preemptions we get

$$l_i < v_i < \frac{v_{i-1}}{(1+\sqrt{k})} < \frac{\mathsf{lstvalue}(I)}{(1+\sqrt{k})^{m-i}} \quad \text{for all } 1 \leq i \leq m-1. \tag{1}$$

T_1 preempted the last order-scheduled task hence (see statement 80 of D^{over})

$$v_1 > (1+\sqrt{k})\{\,\mathsf{recentval}(t_{first\_lst}) + \text{ value(current task at time } t_{first\_lst})\}. \tag{2}$$

Also,

$$t_{first\_lst} - t_{begin} \leq \mathsf{ordervalue}(I) + \mathsf{recentval}(t_{first\_lst}) + \text{value(current task at time } t_{first\_lst}). \tag{3}$$

This holds because the processor is not idle between t_{begin} and t_{first_lst} (as part of $BUSY$) and the right-hand side above represents the sum of the values of all the tasks that were scheduled between t_{begin} and t_{first_lst}. This sum must be greater than or equal to their period of execution by the normalized importance assumption (Assumption 3.2). Inequalities (1)–(3) imply

$$t_{first\_lst} - t_{begin} < \mathsf{ordervalue}(I) + \frac{v_1}{(1+\sqrt{k})} < \mathsf{ordervalue}(I) + \frac{\mathsf{lstvalue}(I)}{(1+\sqrt{k})^m}.$$

We have produced the following bound on the distance between t_{begin} and t_{close}:

[8] Recall that the value density of every task is equal to or greater than 1, by Assumption 3.2 above.

[9] Note that always $l_i \leq v_i$. However, for a task that was abandoned a strict inequality $l_i < v_i$ holds.

$$
\begin{aligned}
t_{close} - t_{begin} &= (t_{first\_lst} - t_{begin}) + (t_{close} - t_{first\_lst}) \\
&= (t_{first\_lst} - t_{begin}) + (l_1 + l_2 + \cdots + l_m) \\
&\leq \mathsf{ordervalue}(I) \\
&\quad + \mathsf{lstvalue}(I) \cdot \left(1 + \frac{1}{(1+\sqrt{k})} + \frac{1}{(1+\sqrt{k})^2} + \cdots + \frac{1}{(1+\sqrt{k})^m}\right) \\
&\leq \mathsf{ordervalue}(I) + \mathsf{lstvalue}(I) \cdot \sum_{i=0}^{\infty} \frac{1}{(1+\sqrt{k})^i} \\
&= \mathsf{ordervalue}(I) + \mathsf{lstvalue}(I) \cdot \left(1 + \frac{1}{\sqrt{k}}\right) \\
&= \mathsf{achievedvalue}(I) + \frac{1}{\sqrt{k}} \cdot \mathsf{lstvalue}(I).
\end{aligned}
$$

The last equality follows from the fact that achievedvalue(I) = ordervalue(I) + lstvalue(I) by definition. □

LEMMA 5.6. *Suppose T_i was abandoned during the interval I. Then*

$$v_i \leq (1+\sqrt{k}) \cdot \mathsf{achievedvalue}(I).$$

Recall that achievedvalue*(I) is the total value obtained during I.*

Proof. Let $I = [t_{begin}, t_{close}]$ be an interval. Define the *Prospective Value* map of I, PV_I, as follows:

$$PV_I(t) = \mathsf{ordervalue}(t) + \mathsf{recentval}(t) + \text{value(current task at time } t) \quad \text{where } t_{begin} \leq t \leq t_{close}.$$

Claim For every interval, $I = [t_{begin}, t_{close}]$,

1. PV_I is monotone nondecreasing.
2. PV_I reaches, at the end of the interval, the total value obtained in I, i.e.,

$$PV_I(t_{close}) = \mathsf{achievedvalue}(I).$$

Note. PV is not a function because it might have several values for one time instance since D^{over} can make several scheduling decisions at one time instance (Assumption 3.2). However, as a map with the ordered sequence of scheduling decisions as its domain, PV_I is a function.

Proof of Claim. There are two cases. The first applies when there are no lst-scheduled tasks in I, the other applies when such tasks exist.

Case 1. Suppose that there are no lst-scheduled tasks in I. Then every task that was scheduled does complete. Let $S(t)$ be the set of tasks that were scheduled (not necessary completed) up to t. One can verify by induction that

$$PV_I(t) = \sum_{T_i \in S(t)} v_i.$$

The reason is that no scheduled task is abandoned hence at each moment a task is either the current task or in Q_privileged or has already completed. At the closing of I all tasks have completed. Hence,

$$PV_I(t_{close}) = \sum_{T_i \in S(t_{close})} v_i = \mathsf{achievedvalue}(I).$$

PV_I is monotone (when there are no lst-scheduled tasks) because $S(t)$ is a monotone increasing set of tasks.

Case 2. Suppose there were lst-scheduled tasks. Assume that the first lst-scheduled task, T_1, was scheduled at time t_{first_lst}. Let t be a time instance just before the scheduling of T_1, then by definition:

$$PV_I(t) = \text{ordervalue}(t) + \text{recentval}(t) + \text{value(current task at time } t)$$

T_1 is scheduled only if

$$v_1 > (1 + \sqrt{k}) \cdot (\text{recentval}(t) + \text{value(current task at time } t))$$

When T_1 is scheduled recentval is set to zero hence we can conclude that

$$\begin{aligned} PV_I(t_{first\_lst}) &= \text{ordervalue}(t_{first\_lst}) + \text{recentval}(t_{first\_lst}) + value(T_1) \\ &= \text{ordervalue}(t) + 0 + value(T_1) \\ &> \text{ordervalue}(t) + (\text{recentval}(t) + \text{value(current task at time } t)) \\ &= PV_I(t) \end{aligned}$$

Thus, PV_I is monotone from t_{begin} to t_{first_lst} (as in the case when there are no lst-scheduled tasks). It is left to show that PV_I continues to be monotone. After t_{first_lst}, PV_I equals to

$$\text{ordervalue}(I) + \text{value(current task at time } t)$$

because recentval remains equal to zero. This is a monotone increasing value since ordervalue(I) is fixed and a task T will preempt the current task only if it has a larger value than the current task's value. In particular if T_i is the last task to be scheduled in I then

$$\begin{aligned} PV_I &= \text{ordervalue}(I) + v_i \\ &= \text{ordervalue}(I) + \text{lstvalue}(I) = \text{achievedvalue}(I). \end{aligned}$$

So, the claim is proved.

End of proof of claim. We return to the proof of Lemma 5.6. There is only one way a task, T_i, can be abandoned at time t:

- T_i reaches its *LST* at t. A latest-start-time interrupt is generated. However, T_i has insufficient value to preempt the task executing at time t.

Hence if T_i was abandoned then

$$\begin{aligned} v_i &< (1 + \sqrt{k}) \cdot \{\text{recentval}(t) + \text{value(current task at time } t)\} \\ &\leq (1 + \sqrt{k}) \cdot PV_I(t), \text{ by definition of } PV, \\ &\leq (1 + \sqrt{k}) \cdot \text{achievedvalue}(I), \text{ by the claim.} \quad \square \end{aligned}$$

LEMMA 5.7. *Suppose T_i was abandoned at time t in $I = [t_{begin}, t_{close}]$. Then*

$$c_i \geq d_i - t_{close}.$$

Proof. A task T_i, can be abandoned at time t only when

- It reaches its *LST* at t. A latest-start-time interrupt is generated. However, the current task is not preempted.

T_i reached its *LST* hence its remaining computation time is $d_i - t$. Also, $t \leq t_{close}$ by assumption. Hence the (initial) computation time of T_i is at least $d_i - t_{close}$. $\square$

5.3. How well can a clairvoyant scheduler do. Given a collection of tasks Γ, our goal is to bound the maximum value that a clairvoyant algorithm can obtain from scheduling Γ. We do it by observing the way D^{over} schedules Γ. From D^{over}'s scheduling we get the partitioning of the tasks to S^0, S^p, and F. We also take notice of the time periods in which the processor was not idle in this scheduling. As defined earlier, the union of these periods is called $BUSY$.

In order to bound the value that can be achieved from scheduling Γ, we will offer the clairvoyant algorithm two gifts that can only improve the value it can obtain. We will show an upper bound on the value the clairvoyant algorithm can get with these gifts hence bounding the value it can achieve from the original collection.

- As a first gift, we will give the clairvoyant algorithm the sum of the values of all tasks in S^p at no cost to it (i.e. it will devote no time to these tasks). Then we will see what the clairvoyant algorithm can achieve on $F \cup S^0$.
- As a second gift, suppose that in addition to the value achieved from scheduling the tasks $F \cup S^0$ the clairvoyant scheduler can get an additional value called *granted value*. The amount of granted value depends on the schedule chosen by the the clairvoyant scheduler: A value density of k will be granted for every period of $BUSY$ that is not used for executing a task. (This is reminscent of farm subsidies for not growing grain.)

The clairvoyant scheduler must consider that scheduling a task might reduce the granted value (since time in $BUSY$ is used). Of course, when this reduction is bigger than the value of a task then the task should not be scheduled. Suppose the clairvoyant algorithm had chosen a scheduling for $F \cup S^0$. We can assume that no task was scheduled entirely during $BUSY$ because the granted value lost would be greater or equal to the value gained from scheduling the task. We have shown that tasks of S^0 can execute only during $BUSY$ hence this leaves only tasks of F that were scheduled partially outside $BUSY$. Executing T results in a gain of $value(T)$, but entails a loss of the granted value for the time that T executed in $BUSY$.

The clairvoyant scheduler has now two options. It can schedule no task during the entire $BUSY$ period and get only (the whole) granted value or it can use some of $BUSY$ in order to schedule some of F tasks. We will show that the maximal possible gain from choosing the second option over the first is bounded by $(1 + \sqrt{k}) \cdot$ achievedvalue. Putting this together will give the desired result (Theorem 5.14).

In the aforementioned scenario the clairvoyant scheduler can achieve (using the gifts) the maximal value of the sum in equation (4) ranging over all possible schedulings[10] of F.

$$\begin{matrix}\text{value obtained from those tasks of } F \\ \text{that were scheduled}\end{matrix} + k \cdot \begin{matrix}\text{length of time in } BUSY \text{ not utilized} \\ \text{to schedule the tasks of } F.\end{matrix} \tag{4}$$

Denote by $C(\cdot)$ the value that a clairvoyant algorithm can achieve from a collection of tasks. We would like to show that $C(F \cup S^0)$ cannot be greater then this maximal value. This will then give us an upper bound on what a clairvoyant algorithm can achieve.

LEMMA 5.8.

$$C(F \cup S^0) \leq \max_{\substack{\text{possible} \\ \text{scheduling of} \\ F}} \left\{ \begin{matrix}\text{value obtained by} \\ \text{scheduling tasks of } F\end{matrix} + k \cdot \begin{matrix}\text{length of time in} \\ BUSY \text{ not utilized by} \\ \text{tasks of } F\end{matrix} \right\}.$$

[10] Suppose a *clairvoyant scheduler* has to schedule a collection of tasks A. We can assume that it schedules a task only if that task eventually completes. Hence the work of a clairvoyant scheduler is first to choose the set of tasks $A' \subseteq A$ that will be scheduled and then to work out the details of the processor allocation among the tasks of A'. We will call all possible selections of A' and processor allocation *a scheduling* of A.

Proof.

$$C(F \cup S^0) = \max \left\{ \begin{array}{c} \textit{value obtained from} \\ \textit{scheduling tasks of } F \end{array} + \begin{array}{c} \textit{value obtained from scheduling tasks of} \\ S^0 \textit{ during the time not used by tasks of } F \end{array} \right\}.$$

S^0 tasks can be scheduled only during $BUSY$ (Lemma 5.4), hence

$$\begin{array}{c} \textit{value obtained from scheduling} \\ \textit{tasks of } F \end{array} + \begin{array}{c} \textit{value obtained from scheduling tasks of } S^0 \\ \textit{during the time not used by tasks of } F \end{array}$$

$$\leq \begin{array}{c} \textit{value obtained by scheduling} \\ F \end{array} + k \cdot \begin{array}{c} \textit{length of time in } BUSY \textit{ not utilized by tasks} \\ \textit{of } F. \end{array}$$

The lemma is proved. □

Suppose a task $T_f \in F$ is scheduled to completion by the clairvoyant algorithm. If T_f executes entirely during $BUSY$ then the left-hand factor of the sum is increased only by v_i which is smaller than or equal to $k \cdot c_i$ while the right-hand factor is decreased by $k \cdot c_i$ giving zero or negative net change. Thus we assume that T_f executes (at least partially) outside $BUSY$.

LEMMA 5.9. *Suppose T_f is abandoned (by D^{over}) at time t_{aban} and that $I = [t_{begin}, t_{close}]$ is the interval in which T_f is abandoned. Then, if T_f is to be executed (by the clairvoyant algorithm) anywhere outside $BUSY$ it must be after t_{close}.*

Proof. $\Delta_f = [r_f, t_{aban}] \cup [t_{aban}, d_f]$. The first portion of Δ_f is contained in $BUSY$ (Lemma 5.4). $[t_{aban}, t_{close}] \subseteq I \subseteq BUSY$, hence if T_i is to be scheduled anywhere outside $BUSY$ it must be after t_{close}.[11] □

Now we are ready to give an upper bound on how much additional value can the clairvoyant algorithm achieve by scheduling tasks of F compared with collecting only the granted value without scheduling any tasks. We make strong use of the fact that when a task T is abandoned during I, T's value cannot be too large with respect to achievedvalue(I).

LEMMA 5.10. *With the above gifts, the total* net *gain obtained by the clairvoyant algorithm from scheduling the tasks abandoned during I is not greater than*

$$(1 + \sqrt{k}) \cdot \text{achievedvalue}(I).$$

Proof. Assume that a clairvoyant scheduler selected a scheduling for the tasks of F, considering the value that can be gained from leaving $BUSY$ periods idle. We can assume that a clairvoyant algorithm executes a task only if this task eventually completes. If the clairvoyant algorithm does not schedule any of the tasks abandoned during I, the lemma is proved. Hence, assume that of all the tasks abandoned in $I = [t_{begin}, t_{close}]$, the clairvoyant scheduler schedules $T_1, T_2, \ldots, T_m$ (in order of completion). These tasks execute for $l_1, l_2, \ldots, l_m$ time *after* t_{close} (hence, maybe outside $BUSY$). We know that all the l_i's are greater than zero (otherwise there is no net gain).

Lemma 5.6 ensures that the biggest possible value of a task to be abandoned during I is $(1 + \sqrt{k}) \cdot \text{achievedvalue}(I)$. If such a task has value density k its execution time is $\frac{(1+\sqrt{k}) \cdot \text{achievedvalue}(I)}{k}$. Denote by L the maximal value of this execution time and the length of l_1

$$L = \max \left\{ \frac{(1 + \sqrt{k}) \cdot \text{achievedvalue}(I)}{k}, l_1 \right\}. \tag{5}$$

[11] Note that parts of $[t_{close}, d_f]$ might be included in $BUSY$ as a new interval may be opened before d_f.

Let j be the index less than to m such that

$$\sum_{i \le j} l_i \le L < \sum_{i \le j} l_i + l_{j+1}.$$

If no such j exists, define j to be m.

First, assume that we have an equality, $\sum_{i \le j} l_i = L$. The $\sum_{i \le j} l_i < L$ case is a little more complicated and will be treated later.

We will show that the net gain from scheduling tasks within a period of L after the end of the interval cannot be greater than $(1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I)$.

- Suppose that in (5), the maximum is the first term. Then the total net gain from $T_1, T_2, \ldots, T_j$ is not greater than

$$k \cdot \sum_{i \le j} l_i = k \cdot L = (1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I). \tag{6}$$

- If on the other hand the second term is maximal in (5) then the value obtained by scheduling T_1 is at most $(1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I)$ (Lemma 5.6).

Now we will show that the net gain from scheduling tasks "after" L is never positive.

Every task T_i that executed at a time of at least L after the end of the interval, where $j < i \le m$, has an execution time c_i of at least $d_i - t_{close}$ (see Lemma 5.7).

$$\begin{aligned} c_i &\ge d_i - t_{close} \\ &\ge \text{"the point at which } T_i \text{ completes (according to the clairvoyant)"} - t_{close} \\ &\ge (t_{close} + \sum_{g \le i} l_g) - t_{close} = \sum_{g \le i} l_g \\ &\ge l_i + \sum_{g \le j} l_g = l_i + L. \end{aligned}$$

For $i > j$, T_i was scheduled by the clairvoyant scheduler but used only l_i time after t_{close}. Hence, T_i executed at least L time before t_{close} that is to say in $BUSY$ by lemma 5.9. The "loss" from scheduling T_i during $BUSY$ is at least $k \cdot L$. The value obtained by scheduling T_i is at most $(1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I)$ (Lemma 5.6). Hence the net gain is less than or equal to

$$\begin{aligned} & (1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I) - k \cdot L \\ \le\ & (1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I) - (1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I) = 0. \end{aligned}$$

We conclude that the clairvoyant algorithm is better off not scheduling any task T_i, $j < i \le m$. Hence, the lemma is proved for the case that $\sum_{i \le j} l_i = L$.

What if L does not equal any of the partial sums? That is, what if $\sum_{i \le j} l_i < L < \sum_{i \le j+1} l_i$? We will augment the total value given to the clairvoyant by some non-negative amount. Then we will show that even with this addition the net gain achieved by the clairvoyant algorithm is bounded by $(1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I)$, hence proving the lemma.

First we will take the value density of T_j to be k. This move can only increase the overall value achieved by the clairvoyant algorithm. We will also "transfer" some execution time (and hence also value) from T_{j+1} to T_j. We will transfer exactly $L - \sum_{i \le j} l_i$ execution time. There will be a nonnegative net increase of $(k - \text{value density}(T_{j+1})) \cdot (L - \sum_{i \le j} l_i)$ in the overall achieved value of the clairvoyant algorithm and we are back in the case of $L = \sum_{i \le j} l_i$. The total net gain from $T_1, \ldots, T_j$ is bounded by $(1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I)$ while the net gain from all other tasks is zero or negative. □

Our strategy thus far has entailed partitioning the problem into what the clairvoyant can obtain with respect to a given interval. We now compute an upper bound for what the clairvoyant algorithm can obtain over all intervals. Note that this may overestimate what the clairvoyant algorithm obtains, because the time periods that the clairvoyant algorithm uses on the tasks of two neighboring intervals may overlap.

COROLLARY 5.11. *With the previous gifts, the total net gain (over the entire execution) obtained by the clairvoyant algorithm from scheduling the tasks of F is not greater than*

$$(1 + \sqrt{k}) \cdot \mathsf{achievedvalue}.$$

Proof. Lemma 5.10 measured the maximum net gain per interval. By construction, each task is accounted for in exactly one interval. Therefore, summing over all intervals we conclude that the total net gain during the entire execution is less than or equals to $(1 + \sqrt{k}) \cdot \mathsf{achievedvalue}$. □

The previous corollary bounds the value that the clairvoyant algorithm could obtain beyond the granted value, which equals $k \cdot BUSY$. Now, we will estimate the granted value (by bounding the length of $BUSY$) to get an upper bound on $C(S^0 \cup F)$.

LEMMA 5.12.

$$C(F \cup S^0) \leq k \cdot (\mathsf{achievedvalue} + \frac{1}{\sqrt{k}} \cdot \mathsf{zerolaxval}) + (1 + \sqrt{k}) \cdot \mathsf{achievedvalue}$$

$$= (k + 1 + \sqrt{k}) \cdot \mathsf{achievedvalue} + \sqrt{k} \cdot \mathsf{zerolaxval}.$$

Proof. Lemma 5.8 shows that $C(S^0 \cup F)$ is bounded by the maximum, ranging over all possible schedulings of the tasks of F, of the following sum:

(value obtained by scheduling F) +

$k \cdot$ (length of time in $BUSY$ not utilized by F tasks).

Corollary 5.11 shows that this sum is less than or equal to

$$(1 + \sqrt{k}) \cdot \mathsf{achievedvalue} + k \cdot BUSY.$$

Lemma 5.5, summed over all intervals, yields

$$BUSY \leq \mathsf{achievedvalue} + \frac{1}{\sqrt{k}} \cdot \mathsf{lstvalue}.$$

$\mathsf{lstvalue}(I) \leq \mathsf{zerolaxval}(I)$ always holds because every task that is lst-scheduled must have completed at its deadline. This implies that

$$BUSY \leq \mathsf{achievedvalue} + \frac{1}{\sqrt{k}} \cdot \mathsf{zerolaxval}.$$

Hence,

$$C(S^0 \cup F) \leq k \cdot \left(\mathsf{achievedvalue} + \frac{1}{\sqrt{k}} \cdot \mathsf{zerolaxval}\right) + (1 + \sqrt{k}) \cdot \mathsf{achievedvalue}$$

$$= (k + 1 + \sqrt{k}) \cdot \mathsf{achievedvalue} + \sqrt{k} \cdot \mathsf{zerolaxval},$$

which proves the lemma. □

We gave the clairvoyant algorithm the value of all tasks in S^p. We also got a bound on $C(S^0 \cup F)$. The following lemma shows that the sum of these two values bounds the value the clairvoyant can get from the entire collection.

LEMMA 5.13.

$$C(F \cup S^0 \cup S^p) \leq C(F \cup S^0) + C(S^p) = C(F \cup S^0) + \sum_{T_i \in S^p} v_i.$$

Proof. The first inequality is due to the fact that $C(\cdot)$ is a sublinear function. The reason is that executing tasks of S^p might interfere with tasks of $F \cup S^0$ and vice versa. Therefore, the value of the union may be less than the sum of the values of the individual sets. D^{over} schedules to completion all the tasks of S^p, hence $C(S^p)$ equals the sum of the values of all these tasks. This yields the desired result. □

Given a collection of tasks Γ, Lemmas 5.12 and 5.13 give an upper bound on the value that the clairvoyant algorithm can obtain from Γ in terms of the value obtained by D^{over} (achievedvalue, zerolaxval, and poslaxval). The next theorem puts these results together.

THEOREM 5.14. *D^{over} has a competitive multiplier of $(1 + \sqrt{k})^2$. That is, D^{over} obtains at least $\frac{1}{(1+\sqrt{k})^2}$ times the value of a clairvoyant algorithm given any task collection Γ.*

Proof. In the notation of the lemmas, we derive from Lemma 5.12 that

$$C(S^0 \cup F) \leq (k + 1 + \sqrt{k}) \cdot \text{achievedvalue} + \sqrt{k} \cdot \text{zerolaxval}.$$

We will bound $\sqrt{k} \cdot$ zerolaxval in the previous equation.

$$\begin{aligned} \sqrt{k} \cdot \text{achievedvalue} &= \sqrt{k} \cdot \text{zerolaxval} + \sqrt{k} \cdot \text{poslaxval} \geq \sqrt{k} \cdot \text{zerolaxval} + \text{poslaxval} \\ &\Rightarrow \sqrt{k} \cdot \text{zerolaxval} \leq \sqrt{k} \cdot \text{achievedvalue} - \text{poslaxval}. \end{aligned}$$

Hence, replacing ($\sqrt{k} \cdot$ zerolaxval) by ($\sqrt{k} \cdot$ achievedvalue $-$ poslaxval) yields

$$\begin{aligned} C(S^0 \cup F) &\leq (k + 1 + \sqrt{k}) \cdot \text{achievedvalue} + \sqrt{k} \cdot \text{achievedvalue} - \text{poslaxval} \\ &= (1 + \sqrt{k})^2 \cdot \text{achievedvalue} - \text{poslaxval}. \end{aligned}$$

By using Lemma 5.13 we get

$$\begin{aligned} C(F \cup S^0 \cup S^p) &\leq C(F \cup S^0) + C(S^p) \\ &= C(F \cup S^0) + \text{poslaxval} \\ &\leq ((1 + \sqrt{k})^2 \cdot \text{achievedvalue} - \text{poslaxval}) + \text{poslaxval} \\ &= (1 + \sqrt{k})^2 \cdot \text{achievedvalue}. \end{aligned}$$

□

5.4. The running complexity of D^{over}. In the previous section, we analyzed the performance of D^{over} in the sense of what value it will achieve from scheduling tasks to completion. In this section we study the cost of executing the scheduling algorithm itself.

THEOREM 5.15. *If n bounds the number of unscheduled tasks in the system at any instant, then each task incurs an $O(\log n)$ amortized cost.*

Proof. D^{over} requires three data structures called Q_privileged, Q_waiting, and Qlst, all of them priority queues, implemented as balanced search trees, e.g., 2–3 trees. They support Insert, Delete, Min, and Dequeue operations, each taking $O(\log n)$ time for a queue with n tasks. The structures share their leaf nodes which represent tasks.

D^{over} consists of a main loop with three "interrupt handlers" within it. The total number of operations is dominated by the number of times each of these handler clauses is executed and the number of data structure operations in each clause.

Suppose a history of m tasks is given. First, let us estimate the number of times each handler clause can be executed. A task during its lifetime causes exactly one task release event and at most one task completion event as well as at most one latest-start-time interrupt event. Hence, while scheduling m tasks the total number of events is bounded by $3m$.

Now we will bound the number of queue operations in each handler clause.

- In the handler for the task release event (statement 54), there is a constant number of queue operations. Hence, this contributes a total of $O(m)$ queue operations during the entire history.
- In the handler for the task completion event (statement 9) there is a constant number of queue operations. Hence, this contributes a total of $O(m)$ queue operations during the entire history.
- In the handler for latest-start-time interrupt event (see statement 76), the number of queue operations is proportional to the number of tasks in Q_privileged plus a constant. (Because the privileged tasks are all inserted into Q_waiting, statement 83.) How many tasks can be in Q_privileged throughout the history? A task can enter Q_privileged only as a result of task release event (statement 64); there are at most m such events. Hence, the total number of tasks in Q_privileged is at most m, which means that the total number of queue operations is $O(m)$ during the entire history.

We conclude that the total number of operations for the entire history is $O(m \log n)$ and the theorem is proved. □

6. Underloaded periods: Conflicting tasks. Intuitively, D^{over} is an optimal scheduler during underloaded periods, because it mimics the *earliest-deadline-first* algorithm during those periods. It gives its nontrivial competitive guarantee during overloaded periods.

To make these statements precise, we must define what underloaded and overloaded mean. Informally, underload means a situation in which all tasks can be scheduled to completion by their deadlines. Such tasks are designated as *conflict-free*. The following algorithm (Fig. 5) gives a precise definition of conflict-free and their antithesis—*conflicting tasks*.[12]

D^{over} schedules to completion all conflict-free tasks (thus, all tasks in an underloaded system) and also obtains at least $\frac{1}{(1+\sqrt{k})^2}$ times the value a clairvoyant algorithm can get from the conflicting tasks. The proof rests on the proof of the competitive guarantee given in this paper and can be found in [9] and [12].

7. Conclusions. This paper has presented an optimal on-line scheduling algorithm for overloaded uniprocessor systems. It is optimal in the sense that it gives the best competitive multiplier possible relative to an off-line scheduler. In fact, the performance guarantee of D^{over} is even stronger: D^{over} schedules to completion all tasks in underloaded periods and achieves at least $\frac{1}{(1+\sqrt{k})^2}$ of the value a clairvoyant algorithm can get during overloaded periods. The model accounts for different value densities and generalizes to soft deadlines [9], [12].

This work leaves many problems open. Here is a small sample.

- This paper assumes that k is given and known in advance. It is interesting to know if this assumption can be relaxed. Recently, in an unpublished result, Schieber [16] devised a variant of D^{over} that gives the optimal guarantee even when k is not known in advance.
- What guarantees can be given for parallel scheduling algorithms? Recently we have found some results in this area, but much remains to be done, because those results are not tight in all cases [9], [10], [13].

[12] Note that the purpose of this algorithm is to define conflicting and conflict-free tasks. No scheduler need ever execute it.

```
1   Function Remove_Conflicts ( Γ ) ;
2
3   if num_of_tasks(Γ) == 1 then
4       return(Γ);
5   endif;
6
7   collection_num_of_tasks :=2;
8   repeat
9       (* Finds a collection of tasks that their combined computation time is
        longer than their combined executable periods *)
10      select a collection of tasks S = T_{i_1}, T_{i_2}, ..., T_{i_{collection_num_of_tasks}} of size
        collection_num_of_tasks such that
        r = Min_{T_i ∈ S}{r_i} and d = Max_{T_i ∈ S}{d_i} and
        c_{i_1} + c_{i_2} + ... + c_{i_{collection_num_of_tasks}} > (d − r);
11      if (such a collection is found) then
12          mark all the tasks in S as conflicting tasks;
13          create a task T with release time r and deadline d
            and with no slack time;
14          (* T is an aggregated task *)
15          return( remove_conflicts( Γ − S + {T}));
            (* Start again with the new collection of tasks. The new collection has
16          a smaller number of tasks. When the recursive calls reach the bottom
            of the recursion (that is when Γ has no conflicting tasks) the result is
            propagated upwards (tail recursion). *)
17      else
18          collection_num_of_tasks := collection_num_of_tasks + 1;
19      endif;
20  until collection_num_of_tasks > num_of_tasks(Γ);
21  return(Γ) (*In case that no conflict was found *)
```

FIG. 5. *The remove conflicts algorithm.*

- What guarantees can be given when tasks are not independent, e.g., for systems with locks or precedence constraints?
- In practice, real-time systems have some periodic critical tasks and other less critical tasks which may be aperiodic. A typical solution (as taken in the Spring Kernel, for example, in [19]) is to devote certain intervals to the critical tasks and to allow the less critical tasks to run during the rest of the time. D^{over} gives its usual guarantee with respect to the less critical tasks in this situation. (The accounting is a little more difficult since useful time has "holes" in it which correspond to subintervals allocated to critical tasks.) An unanswered question follows: What is a good competitive algorithm that can take advantage of the cases when a given critical task executes in less time than is allocated for it? We suspect the competitive guarantee may be worse, since the clairvoyant algorithm might then execute a task that D^{over} had prematurely abandoned.
- In general, the question of proof tools for such systems is open. We believe that the techniques in §5.3 will prove to be very useful.

Acknowledgments. The authors thank Marc Donner, Bud Mishra, and Doug Locke for many inspiring discussions.

REFERENCES

[1] N. AUDSLEY AND A. BURNS, *Real time system scheduling*, Computer Science Department Technical Report YCS134, York University, UK, 1990.

[2] T. P. BAKER AND A. SHAW, *The cyclic executive model and Ada*, The Journal of Real-Time Systems, 1 (1989), pp. 7–25,

[3] S. BARUAH, G. KOREN, D. MAO, B. MISHRA, A. RAGHUNATHAN, L. ROSIER, D. SHASHA, AND F. WANG, *On the competitiveness of on-line task real-time task scheduling*, The Journal of Real-Time Systems, 4 (1992), pp. 124–144. Conference version appeared in Proceedings of the 12th Real- Time Systems Symposium, IEEE Computer Society Press, pp. 106–115, San Antonio, Texas, Dec. 1991.

[4] S. BARUAH, G. KOREN, B. MISHRA, A. RAGHUNATHAN, L. ROSIER, AND D. SHASHA, *On-line scheduling in the presence of overload*, in Proceedings of the 32nd Annual Symposium on the Foundations of Computer Science, San Juan, Puerto Rico, Oct. 1991, IEEE Computer Society Press, pp. 101–110.

[5] M. L. DERTOUZOS, *Control robotics: the procedural control of physical processes*, in Proceedings IFIF Congress, 1974, pp. 807–813.

[6] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: a guide to the theory of NP-Completeness*, W. H. Freeman and Co., New York, 1979 .

[7] J. R. HARITSA, M. J. CAREY, AND M. LIVNY, *On being optimistic about real-time constraints*, in Proceedings of the PODS Conference, Nashville, TN, Apr. 1990, ACM, pp. 331–343.

[8] A. KARLIN, M. MANASSE, L. RUDOLPH, AND D. SLEATOR, *Competitive snoopy caching*, Algorithmica, 3 (1988), pp. 79–119.

[9] G. KOREN, *Competitive On-Line Scheduling for Overloaded Real-Time Systems*, Ph.D. thesis, Computer Science Department, Courant Institute, New York University, New York, 1993.

[10] G. KOREN AND D. SHASHA, *MOCA: A multiprocessor on-line competitive algorithm for real-time system scheduling*, Theoret. Comput. Sci., 128 (1994), pp. 75–97.

[11] ———, *An optimal scheduling algorithm with a competitive factor for real-time systems*, Computer Science Department Technical Report 572, Courant Institute, New York University, New York, July 1991.

[12] ———, *D-over: An optimal on-line scheduling algorithm for overloaded real-time systems*, Computer Science Department Technical Report 594, Courant Institute, New York University, New York, 1992. Also, Technical Report 138, INRIA, Rocquencourt, France, 1992.

[13] G. KOREN, D. SHASHA, AND S.-C. HUANG, *MOCA: A multiprocessor on-line competitive algorithm for real-time system scheduling*, in Proceedings of the 14th Real-Time Systems Symposium, Raleigh-Durham, NC, Dec. 1993, IEEE Computer Society Press, pp. 172–181.

[14] C. D. LOCKE, *Best-Effort Decision Making for Real-Time Scheduling*, Ph.D. thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1986.

[15] A. K.-L. MOK, *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Boston, MA, 1983.

[16] B. SCHIEBER, *private communication*. I.B.M, T.J. Watson Research Center, Yorktown Heights, NY, 1992.

[17] L. SHA, J. P. LEHOCZKY, AND R. RAJKUMAR, *Solutions for some Practical Problems in prioritized preemptive scheduling*, in Proceedings of the 7th Real-Time Systems Symposium, New Orleans, LA, Dec. 1986, IEEE Computer Society Press, pp. 181–191.

[18] D. SLEATOR AND R. TARJAN, *Amortized efficiency of list update and paging rules*, Comm. ACM, 28 (1985), pp. 202–208.

[19] J. A. STANKOVIC AND K. RAMAMRITHAM, *The spring kernel: A new paradigm for real-time systems*, IEEE software, (1991), pp. 62–72.

[20] F. WANG AND D. MAO, *Worst case analysis for on-line scheduling in real-time systems*, Department of Computer and Information Science, Technical Report 91-54, University of Massachusetts, Amherst, MA, 1991.

SIAM J. COMPUT.
Vol. 24, No. 2, pp. 340–356, April 1995

010

ORDERED AND UNORDERED TREE INCLUSION*

PEKKA KILPELÄINEN† AND HEIKKI MANNILA†

Abstract. The following tree-matching problem is considered: Given labeled trees P and T, can P be obtained from T by deleting nodes? Deleting a node u entails removing all edges incident to u and, if u has a parent v, replacing the edge from v to u by edges from v to the children of u. The problem is motivated by the study of query languages for structured text databases. Simple solutions to this problem require exponential time. For ordered trees an algorithm is presented that requires $O(|P||T|)$ time and space. The corresponding problem for unordered trees is also considered and a proof of its NP-completeness is given. An algorithm is presented for the unordered problem. This algorithm works in $O(|P||T|)$ time if the out-degrees of the nodes in P are bounded by a constant, and in polynomial time if they are $O(\log |T|)$.

Key words. trees, pattern matching, dynamic programming, NP-completeness

AMS subject classifications. 68Q20, 68Q25, 68R10

1. Introduction and motivation.

Let T be a tree and u be a node in T with parent node v. Denote by $delete(T, u)$ the tree obtained from T by removing the node u. The children of u become children of v. (See Fig. 1.)

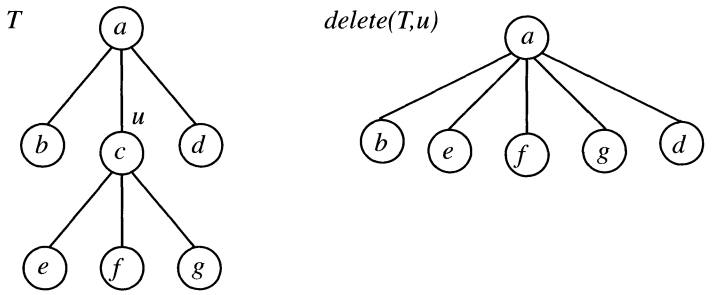


FIG. 1. *The effect of removing the node u from the tree T.*

We consider the following problem: Given two trees P and T, called the *pattern* and the *target*, can we obtain pattern P by deleting some nodes from target T? That is, is there a sequence $u_1, \ldots, u_k$ of nodes such that for

$$\begin{aligned} T_0 &= T \text{ and} \\ T_{i+1} &= delete(T_i, u_{i+1}) \text{ for } i = 0, \ldots, k-1 , \end{aligned}$$

we have $T_k = P$. If this is the case, what are the nodes in T that are (not) deleted by the sequence? We call this problem the *tree inclusion problem.*

Our specific interest in the tree inclusion problem comes from its applicability in structured text databases. A structured text database can be considered as a collection of parse trees that represent the structure of the stored documents [6], [11], [24]. Tree inclusion has been suggested as a primitive for expressing queries on structured documents [17] and as a means of retrieving information from structured documents [13].

*Received by the editors August 19, 1991; accepted for publication (in revised form) October 22, 1993. A preliminary version of this article has appeared in *Proceedings of the International Joint Conference on the Theory and Practice of Software Development*, Vol. 1, Springer-Verlag, New York, 1991, pp. 202–214.

†Department of Computer Science, University of Helsinki, P.O. Box 26 (Teollisuuskatu 23), SF-00014 University of Helsinki, Finland (kilpelai@cs.helsinki.fi and mannila@cs.helsinki.fi).

As an example, consider querying grammatical structures. Figure 2 shows the parse tree of a natural language sentence. One might want to locate, say, those sentences that include a verb phrase containing the verb "holds" and the noun "cat" and any adverb, in this order. These are exactly the sentences whose parse tree can be transformed to the tree in Fig. 3 by deleting nodes.

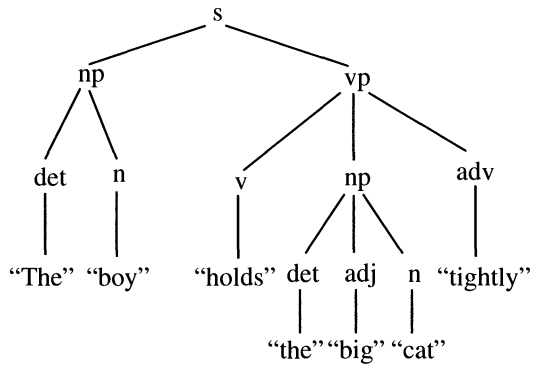


FIG. 2. *The parse tree of a sentence.*

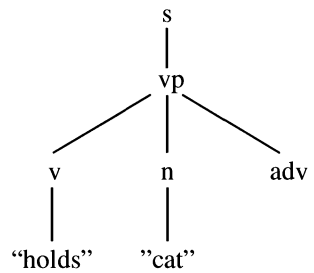


FIG. 3. *An included tree of the parse tree.*

The tree inclusion problem resembles *tree pattern matching* [8], [16], [4], [19], which has several applications. In that problem one wants to find out whether one can obtain a tree that has P as a subtree by removing complete sets of descendant nodes from T. The resemblance of the problems apparently does not extend to the algorithms. For tree pattern matching, the trivial algorithm works in time $O(|P||T|)$, where $|P|$ is the number of nodes in tree P, and $|T|$ is the number of nodes in tree T. The reason why tree pattern matching is a fairly simple problem is that there are only $|T|$ possible places where P can be matched in T. Hence the trivial algorithm can simply test all the possibilities. The $O(|P||T|)$ bound was not broken unitl recently [16], [4].

For the tree inclusion problem, there are exponentially many ways to obtain the included tree. Thus it is not feasible to check all the possibilities.

In this paper, we analyze the computational complexity of the tree inclusion problem. We give a polynomial time algorithm for solving the tree inclusion problem for ordered trees, i.e., when the ordering of siblings has significance. The algorithm is based on dynamic programming and it works in time $O(|P||T|)$.

The ordering among siblings is crucial for solving the tree inclusion problem efficiently. In the case of unordered trees, the problem is NP-complete. This result follows from the known NP-completeness of MINOR CONTAINMENT [9], [21]. To our knowledge, the

proof has not been published. We present a proof that is based on a direct reduction from SATISFIABILITY.

We also present an algorithm for solving the tree inclusion problem on unordered trees. The algorithm may require time that is exponential, but only in the width of the pattern. This means that if the out-degrees of the pattern nodes are $O(\log |T|)$, the unordered tree inclusion problem is solvable in polynomial time. This would usually be the case if we think P as a query and T as a large database [13].

The rest of this paper is organized as follows. We start by giving definitions and some properties of tree inclusion in §2. In §3 we relate the tree inclusion problem and our results to previous research, then we concentrate on solving the ordered tree inclusion problem. In §4 we develop the idea of *left embeddings*, which is the basis of our algorithms. Section 5 presents the basic algorithm. This algorithm may repeat the same computations many times, which leads to an exponential behavior. In §6 we develop the algorithm further. The use of a table for storing the results of subcomputations leads to an algorithm requiring $O(|P||T|)$ running time and storage. In §7 we consider the tree inclusion problem with unordered trees and give a proof of its NP-completeness. In §8 we present and analyze an algorithm for solving the unordered tree inclusion problem. Section 9 is a short conclusion.

2. Forest inclusion. We concentrate on labeled trees that are *ordered*, i.e., the order between siblings is significant. Ordered labeled trees appear in various fields, including programming language implementation, natural language processing, and molecular biology.

Technically, it is convenient to consider a slight generalization of trees, namely forests. A *forest* is a finite ordered sequence of disjoint finite trees. A *tree* T consists of a specially designated node $root(T)$ called the *root* of the tree, and a forest $\langle T_1, \ldots, T_k \rangle$, where $k \geq 0$. The trees $T_1, \ldots, T_k$ are the *subtrees* of the root of T or the *immediate subtrees* of tree T, and k is the *out-degree* of the root of T. The roots of the trees $T_1, \ldots, T_k$ are the *children* of the root of T and *siblings* of each other. The root is an *ancestor* of all the nodes in its subtrees, and the nodes in the subtrees are *descendants* of the root. The set of descendants of a node u is denoted by $desc(u)$. A *leaf* is a node with an empty set of descendants.

Sometimes we treat a tree T as the forest $\langle T \rangle$. We may also denote the set of nodes in a forest F by F. For example, if we speak of functions from a forest F to a forest G, we mean functions mapping *nodes* of F onto *nodes* of G. The *size* of a forest F, denoted by $|F|$, is the number of nodes in F.

The restriction of a forest F to a node u with its descendants is called the *subtree of F rooted by u*, and it is denoted by $F[u]$.

Let $F = \langle T_1, \ldots, T_k \rangle$ be a forest. The *preorder of a forest* F is the order of the nodes visited during a preorder traversal. A *preorder traversal of a forest* $\langle T_1, \ldots, T_k \rangle$ is as follows: Traverse the trees $T_1, \ldots, T_k$ in ascending order of the indices in preorder. To traverse a tree in preorder, first visit the root and then traverse the forest of its subtrees in preorder. The *postorder* is defined similarly, except that in a postorder traversal the root is visited *after* traversing the forest of its subtrees in postorder. The preorder and postorder numbers of a node u are denoted by $pre(u)$ and $post(u)$.

The following lemma binds together the ancestorship and the preorder and postorder.

LEMMA 2.1. *Let u and v be nodes in a forest F. Then u is an ancestor of v if and only if $pre(u) < pre(v)$ and $post(v) < post(u)$.*

Proof. See Exercise 2.3.2-20 in [15]. □

Let N be a set of nodes. A *labeling* for N is a function from N to some alphabet.

DEFINITION 2.2. *Let F and F' be forests, N and N' be the sets of their nodes, and let* label *be a labeling for $N \cup N'$. An injective function $f : N \to N'$ is an* embedding of F in F', *if for all nodes $u, v \in N$*

1. f preserves labels, *i.e.*, $label(f(u)) = label(u)$,
2. f preserves ancestors, *i.e.*, $f(u) \in desc(f(v))$ *if and only if* $u \in desc(v)$, *and*
3. f preserves the left-to-right order *of nodes.*

If there is an embedding of F *in* F', *we say that* F' includes F, *denoted* $F \sqsubseteq F'$, *and that* F *is an* included forest (or tree) *of* F'. *Forests* F *and* F' *are* isomorphic *if there is a bijective embedding of* F *in* F'.

If the second condition in the above definition is satisfied, the condition on the left-to-right order can be stated as

$$pre(u) < pre(v) \text{ if and only if } pre(f(u)) < pre(f(v)) ,$$

or, by Lemma 2.1, equivalently as

$$post(u) < post(v) \text{ if and only if } post(f(u)) < post(f(v)) .$$

EXAMPLE 2.3. *The included forests of the forest* $\langle a(b, c)\rangle$ *are* $\langle\rangle$, $\langle a\rangle$, $\langle b\rangle$, $\langle c\rangle$, $\langle b, c\rangle$, $\langle a(b)\rangle$, $\langle a(c)\rangle$, *and* $\langle a(b, c)\rangle$.

LEMMA 2.4. *The relation* $\sqsubseteq$ *is a partial ordering* (*up to isomorphism*), *i.e., it is reflexive, transitive, and antisymmetric.*

The next lemma justifies the formulation of the tree inclusion problem given in the introduction.

LEMMA 2.5. *A forest* G *includes a forest* F *if and only if a forest isomorphic to* F *can be obtained from* G *by deleting nodes.* (*The deletion of a node* v *from* G *replaces the tree* $G[v]$ *by the subtrees of* v.)

Proof. If u and v are nodes of G, the removal of a third node does not change the relative ordering of u and v. □

A forest may have exponentially many included forests, as shown in the next lemma.

LEMMA 2.6. *A forest* F *may have* $\binom{|F|}{m}$ *nonisomorphic included forests of size* m.

Proof. If every node in F has a different label, we get a nonisomorphic included forest of size m for every possible selection of $|F| - m$ nodes to be deleted. The number of such selections is $\binom{|F|}{m}$. □

EXAMPLE 2.7. *The forest* $\langle a(x, b(y(d, e), f), c)\rangle$ *has* 8 *nodes with different labels. Therefore it has* 256 *nonisomorphic included forests. If the labels are similar, the number of nonisomorphic included forests is smaller. For example, the forest* $\langle a(a, a(a(a, a), a), a)\rangle$ *has same size and form as the previous one, but it has only* 68 *nonisomorphic included forests.*

3. Related research. The ordered tree inclusion problem can be found in Exercise 2.3.2-22 of [15]. Knuth gives a sufficient condition for the existence of an embedding.

Tree inclusion can be considered to be a special case of the editing distance problem for trees [25], [27]. In [27], Zhang and Shasha give an algorithm for computing the edit distance. It can also be used for solving the ordered tree-inclusion problem. Their algorithm requires time

$$O(|P| \cdot |T| \cdot min(depth(P), leaves(P)) \cdot min(depth(T), leaves(T))) .$$

Thus their solution is slower than ours by a factor of

$$min(depth(P), leaves(P)) \cdot min(depth(T), leaves(T)) .$$

Shasha and Zhang have recently presented new sequential and parallel algorithms for the editing distance problem with unit cost edit operations [22].

The tree inclusion problem is a generalization of the *subsequence problem* for strings. ("Can the string x be obtained from the string y by deleting characters?") The subsequence problem can be solved in linear time by a straightforward scan. On the other hand, tree inclusion problem is a special case of the MINOR CONTAINMENT problem for graphs [21], [9]. In that problem, given two graphs $G = (V, E)$ and $H = (U, F)$, one has to decide whether G contains H as a *minor*, i.e., is there a subgraph of G that can be converted to H by a sequence of "contractions." In a contraction, two adjacent vertices and an edge between them are replaced by a single new vertex. All the other edges previously incident on either contracted vertex are then viewed incident to the new vertex.

MINOR CONTAINMENT is known to be NP-complete even for free trees, when both H and G are given as inputs.[1] This implies that MINOR CONTAINMENT is NP-complete also for rooted trees or, using our terminology, that tree inclusion is NP-complete for unordered trees. Recently Matoušek and Thomas [18] have independently published a proof of NP-completeness of unordered tree inclusion.

This classification suggests some further problems. It is possible to compute a maximal common subsequence for two strings in quadratic time and in linear space [7]. The corresponding problem of finding a maximal common included tree of two trees can be solved by Zhang and Shasha's editing distance algorithm [27]. Our algorithm specialized for the tree embedding problem is simpler and more efficient than Zhang and Shasha's algorithm. It is not clear yet if the algorithm also can be extended to solve the largest common included tree problem.

A related problem is the *tree pattern matching problem.* (See, e.g., [8], [19].) In tree pattern matching one is given a pattern tree (of m nodes), possibly with variables standing for arbitrary subtrees, and a subject tree (of n nodes). The problem is to locate the subtrees of the subject tree that are isomorphic to some tree presented by the pattern. The $O(mn)$ time bound of the naive algorithm has been difficult to improve for the general case. A recent paper of Kosaraju [16] presents an $O(nm^{0.75}\ polylog(m))$ algorithm. Dubiner, Galil, and Magen improve this result in their paper [4] by presenting an $O(n\sqrt{m}\ polylog(m))$ algorithm.

Checking whether an unordered tree is a subgraph of another one can be done in time $O(m^{3/2}n)$ by using bipartite matching [20], [26].

A unified treatment of related tree matching problems with some further variations is presented in [14].

4. Left embeddings. In this section, we develop concepts that help us efficiently solve the ordered tree inclusion problem. Throughout the rest of the paper, let *label* be a labeling for the nodes of the trees.

We concentrate on searching *root-preserving embeddings.*

DEFINITION 4.1. *Let P and T be trees. A* root-preserving embedding of P in T *is an embedding f of P in T such that $f(root(P)) = root(T)$. If there is a root-preserving embedding of P in T, we say that the root node of T is an* occurrence *of P.*

The following simple lemma states that we do not lose generality by restricting to root-preserving embeddings.

LEMMA 4.2. *Let F and G be forests, and let P and T be two trees whose roots are labeled by the same symbol, such that the immediate subtrees of P form forest F and the immediate subtrees of T form forest G. Then $F \sqsubseteq G$ if and only if there is a root-preserving embedding of P in T.*

[1] For every *fixed* planar graph (and therefore for every fixed tree and forest) H there is a polynomial time algorithm for testing whether H is a minor of a given graph G [21], [9].

There may be exponentially many ways to embed the subtrees of a tree P in the subtrees of a tree T. In order to limit the search among these embeddings, we develop algorithms that search for a root-preserving embedding of P in T by processing the subtrees of P from left to right and trying to embed them as deep and as left as possible in T.

In order to discuss the relationship between images of sibling nodes in an embedding, we define the sets of *right and left relatives* of a node.

DEFINITION 4.3. *Let F be a forest, N be the set of its nodes, and v be a node in F. The set of* right relatives *of v is defined by*

$$rr(v) = \{x \in N \mid pre(v) < pre(x) \wedge post(v) < post(x)\} ,$$

i.e., the right relatives of v are those nodes that follow v both in preorder and in postorder. The set of left relatives *of node v, denoted by $lr(v)$, is the set of nodes that precede v both in preorder and in postorder. (See Fig. 4.)*

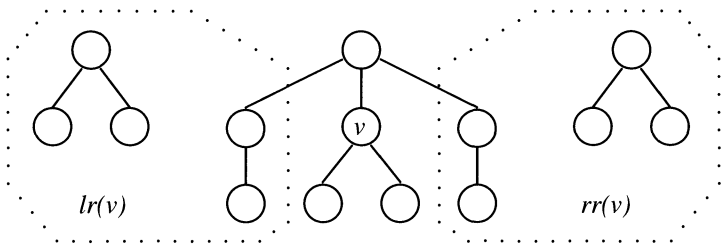


FIG. 4. *The left and right relatives of node v.*

The crucial observation is that an embedding that maps a node u to a node v can map right siblings of u to the right relatives of v only.

The next lemma can be used as a tool with Lemma 2.1 to derive the two lemmas following it.

LEMMA 4.4. *Let u, v, and x be three nodes in a forest. Then it is not possible that both*

$$\begin{array}{ccccc} pre(u) & < & pre(v) & < & pre(x) \quad \text{and} \\ post(x) & < & post(u) & < & post(v) \end{array}$$

hold.

Proof. The above conditions imply by Lemma 2.1 that u and v are ancestors of x but neither of them is an ancestor of the other, which is not possible in a forest. □

The next simple lemma states that the descendants of a right relative are also right relatives.

LEMMA 4.5. *Let v and x be nodes, and assume $x \in rr(v)$. Then $desc(x) \subset rr(v)$.*

The next lemma states that the right relatives of a node v are contained in the right relatives of the nodes that precede v in postorder. This fact is the justification for the strategy of embedding the trees as early as possible in postorder, when searching for an embedding of a forest.

LEMMA 4.6. *Let v and x be nodes in a forest. If $post(v) \leq post(x)$, then $rr(x) \subseteq rr(v)$.*

The next definition gives a name for the embeddings that are searched for by our algorithms.

DEFINITION 4.7. *Let $F = \langle T_1, \ldots, T_k \rangle$ and G be forests, and let $\mathcal{E}$ be a collection of embeddings of F in G. An embedding $f \in \mathcal{E}$ is a* left embedding *of $\mathcal{E}$ if for every $g \in \mathcal{E}$*

$$post(f(root(T_k))) \leq post(g(root(T_k))) .$$

A left embedding of the set of all embeddings of F in G is a left embedding *of F in G.*

It is obvious that every finite nonempty set of embeddings has at least one left embedding. Therefore the following theorem holds, allowing us to restrict to left embeddings when testing for the existence of an embedding.

THEOREM 4.8. *Let F and G be forests. There is an embedding of F in G if and only if there is a left embedding of F in G.*

The next theorem presents a method to build left embeddings by proceeding from left to right. The method is applied in Algorithms 5.1 and 6.1.

THEOREM 4.9. *Let $F = \langle P_1, \ldots, P_k \rangle$, where $k \geq 2$, and G be forests, and let f be a left embedding of $F_1 = \langle P_1, \ldots, P_i \rangle$ in G, where $1 \leq i < k$. Let F_2 be the forest $\langle P_{i+1}, \ldots, P_k \rangle$ and let $\mathcal{E}$ be the set of such embeddings g of F_2 in G that $g(root(P_{i+1})) \in rr(f(root(P_i)))$. Then the following items hold:*

1. *If $\mathcal{E}$ is empty, there is no embedding of F in G.*
2. *If $\mathcal{E}$ is nonempty and g is a left embedding of $\mathcal{E}$, then $f \cup g$ is a left embedding of F in G.*

Proof.

1. Assume contrapositively that g is an embedding of F in G. Let g_1 and g_2 be the restrictions of g in F_1 and F_2, respectively. Then $g_2(root(P_{i+1})) \in rr(g_1(root(P_i)))$, and $post(f(root(P_i))) \leq post(g_1(root(P_i)))$, since f is a left embedding of F_1 in G. Therefore

$$g_2(root(P_{i+1})) \in rr(f(root(P_i)))$$

by Lemma 4.6, which means that $g_2 \in \mathcal{E}$.

2. First we show that $f \cup g$ is an embedding of F in G. It is sufficient to show that $f \cup g$ preserves the relative order of any two nodes x of F_1 and y of F_2. Now $root(P_i)$ is the last node of forest F_1 in postorder. Therefore we have by Lemma 4.6 that $root(P_j) \in rr(x)$ for all $i < j \leq k$, and Lemma 4.5 states that $y \in rr(x)$ also when y is not the root of any of the trees $P_{i+1}, \ldots, P_k$. Next we can see that $g(y) \in rr(f(x))$ by applying the same argument to the images of the above nodes.

Next we show that $f \cup g$ is a left embedding of F in G. For this, let h be any embedding of F in G, and let h_1 and h_2 be the restrictions of h to F_1 and F_2, respectively. Now we have that $h_2 \in \mathcal{E}$, and therefore $post(g(root(P_k))) \leq post(h_2(root(P_k)))$. This entails the claim since $g(root(P_k)) = (f \cup g)(root(P_k))$ and $h_2(root(P_k)) = h(root(P_k))$. □

5. The basic algorithm. Now we are ready to give an algorithm for testing whether there is a root-preserving embedding of a pattern tree P in a target tree T. Let $P_1, \ldots, P_k$ be the immediate subtrees of P. First, our algorithm searches the image $f(root(P_1))$ of $root(P_1)$ under a left embedding f of $\langle P_1 \rangle$ in the immediate subtrees of tree T, if such an embedding exists. The algorithm uses a pointer p for traversing the descendants of $root(T)$. After finding a left embedding f for the forest $\langle P_1, \ldots, P_i \rangle$, p points at node $f(root(P_i))$. In order to extend f to a left embedding of $\langle P_1, \ldots, P_{i+1} \rangle$ in the subtrees of $root(T)$, we try to find the closest right relative x of p in postorder, such that x is an occurrence of P_{i+1} and a descendant of $root(T)$.

The algorithm refers to nodes by their postorder numbers. The minimum of a set of nodes is the first node of the set in postorder. The algorithm also refers to an auxiliary target node 0, which precedes every other node of tree T in both preorder and postorder. This means that all other target nodes are right relatives of node 0. We denote the size of T by n. The root of T being the last target node in postorder is then n.

ALGORITHM 5.1. *Testing whether a tree T includes a tree P.*
Input: *Trees P and T.*
Output: **true** *if and only if there is a root preserving embedding of P in T.*
Method: *Call emb(root(P), root(T));*

```
1.  function emb(u, v);
2.      if label(u) ≠ label(v) then return false;
3.      else
4.          let u_1, ..., u_k be the children of u; (if u is a leaf, k = 0)
5.          p := min(desc(v) ∪ {n + 1}) − 1;
6.          comment: p is the predecessor of desc(v), or n if v is a leaf;
7.          i := 0;
8.          while i < k and p < v do
9.              p := min({x ∈ rr(p) | emb(u_{i+1}, x)} ∪ {n + 1});
10.             comment: p is the next occurrence of P[u_{i+1}],
                        or n + 1 if there is none;
11.             if p ∈ desc(v) then
12.             comment: p = f(u_{i+1}) for a left embedding f
                        of ⟨P[u_1], ..., P[u_{i+1}]⟩ in desc(v);
13.                 i := i + 1;
14.             fi;
15.         od;
16.         if i = k then return true;
17.         else return false;
18.         fi;
19.     fi;
20. end.
```

The loop on lines 8–15 tests whether there is a left embedding of the forest $\langle P[u_1], \ldots, P[u_k]\rangle$ in the forest of subtrees of v. This is, by Theorem 4.8 and Lemma 4.2, equivalent to testing whether there is a root-preserving ordered embedding of $P[u]$ in $T[v]$, because on these lines $label(u) = label(v)$. The test is successful if and only if all the subtrees of u can be embedded in the loop, i.e., if and only if $i = k$ on line 16.

The test is obviously correct if the pattern node u is a leaf. Let node u have children. If the target node v is a leaf, p gets value n on line 5 of the algorithm, which prevents the execution of lines 8–16. This is consistent with the fact that in this case $P[u]$ cannot be embedded in $T[v]$. Otherwise, if target node v has descendants, p gets the value $min(desc(v)) - 1$, which is the closest left relative of v. The postorder numbers of the descendants of v then, are $\{p + 1, p + 2, \ldots, v - 1\}$. By Lemma 4.5, $desc(v) \subset rr(p)$. Then, if $P[u_1]$ has occurrences in $desc(v)$, the first execution of line 9 finds in p the first of them in postorder. The correctness of the remaining executions of the loop follows from Theorem 4.9.

The primitive operations of moving to the first descendant node or to the next or previous node in postorder can be performed in constant time, after a linear time preprocessing of tree T. Note that the tests $p \in desc(v)$ and $x \in rr(p)$ can be realized as simple comparisons of preorder and postorder numbers. (See Lemma 2.1 and Definition 4.3.)

The algorithm above may still need exponential time in the size of the trees. Consider trees $P_n = r(a(a(\ldots a(a(b))\ldots)))$ (a b-leaf with n a-ancestors below root r) and $T_n = r(a(a(\ldots a(\ldots a(a)\ldots), b)\ldots)))$ (a chain of $2n$ a-nodes below root r; the nth a has also a b-leaf as a child). The algorithm would try in the recursive calls on line 9 a total of $\binom{2n}{n}$ embeddings before finding the right images for the a-nodes and an embedding for the whole tree P_n.

6. A dynamic programming solution. The previous algorithm may repeat the same computations an exponential number of times in the worst case. To avoid this, we use an $m \times n$ table ($m = |P|, n = |T|$) to store results of subcomputations. As before, we test for the existence of a root-preserving embedding of $P[u]$ in $T[v]$ by trying to find a left embedding of u's subtrees in v's subtrees. The key is to organize the evaluation so that the step that extends the left embedding of $\langle P_1, \ldots, P_i \rangle$ into an embedding of $\langle P_1, \ldots, P_{i+1} \rangle$ can either find the correct occurrence of P_{i+1} or decide that there is none in *constant time*.

Let us define a table e having rows $1, \ldots, m$ and columns $0, \ldots, n-1$. As before, we refer to the nodes of P and T by their postorder numbers; the numbers of P are used as row indices of the table, and the numbers of T are used as column indices and contents of the table. Denote by $R(P, T)$ the collection of root-preserving embeddings of a tree P in a tree T. We compute into table e values ($u \in P$, $v \in \{0, \ldots, n-1\}$)

$$e(u, v) = min(\{x \in rr(v) \mid \exists f \in R(P[u], T[x])\} \cup \{n+1\}) . \tag{1}$$

That is, $e(u, v)$ contains the closest right relative of target node v that is an occurrence of $P[u]$, or $n+1$ meaning that $P[u]$ has no occurrences among the right relatives of v. The result of the computation can be found on row m (i.e., *root*(P)) of the table. Pattern P can be embedded in T if and only if $e(m, 0) \leq n$, and every $v \in T$ that appears on row m of table e is an occurrence of P.

The new algorithm uses pointer p in the same way as Algorithm 5.1 did. When a root-preserving embedding of $P[u]$ in $T[v]$ is found, another pointer q is used for writing value v into $e(u, q)$ for those nodes q for which v is the appropriate value along (1). Since those nodes q are left relatives of node v, columns $0 \ldots n-1$ suffice for the table because node n, i.e., the root of the target, has no right relatives.

ALGORITHM 6.1. *Testing whether a tree T includes a tree P, dynamic programming version.*
Input: *Trees P and T ($|P| = m$, $|T| = n$).*
Output: *Table e filled so that for all $u \in P$ and $v = 0, \ldots, n-1$*

$$e(u, v) = \min(\{x \in rr(v) \mid \exists f \in R(P[u], T[x])\} \cup \{n+1\}) .$$

Method:

```
1.  for u := 1, ..., m do
2.      comment: Initialize row u of e;
3.      for v := 0, ..., n - 1 do e(u, v) := n + 1; od;
4.      Let u_1, ..., u_k be the children of u; (if u is a leaf, k = 0)
5.      q := 0;
6.      for v := 1, ..., n do
7.          if label(v) = label(u) then
8.              p := min(desc(v) ∪ {n + 1}) - 1;
9.              i := 0;
10.             while i < k and p < v do
11.                 p := e(u_{i+1}, p);
12.                 if p ∈ desc(v) then i := i + 1; fi;
13.             od;
14.             if i = k then
15.                 while q ∈ lr(v) do
16.                     e(u, q) := v;
17.                     q := q + 1;
18.                 od;
```

19. **fi**;
20. **fi**;
21. **od**;
22. **od**.

As an example, consider how Algorithm 6.1 finds the embedding of a tree $P = a(c, e)$ in a tree $T = a(b(c), a(b(d), a(b(e))))$. The trees and the result of the computation can be seen in Fig. 5. Each column of table e is shown to the right of the corresponding target node.

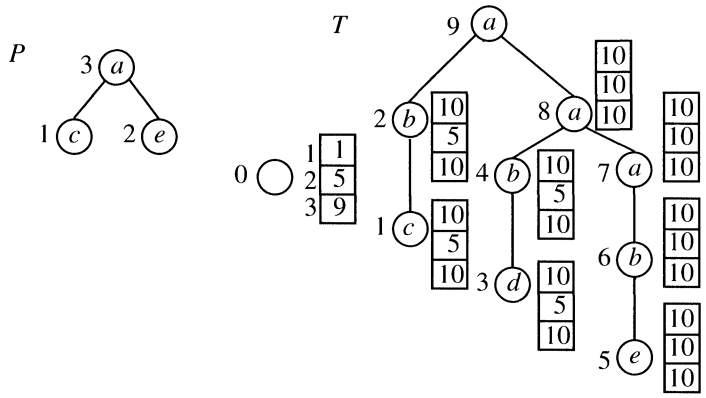


FIG. 5. *The result of applying the algorithm to trees P and T.*

First, $u = 1$, the leaf of P labeled by c, and $v = 1$, the similar leaf of T. Since the labels match and u has no children ($i = k = 0$), we have an embedding. The value of $v = 1$ is written into $e(1, 0)$ only, since 0 is the only left relative of v. After that, no more matching labels are found for $u = 1$ in nodes $v = 2, \ldots, 9$ of T.

Next $u = 2$, the second leaf of P. The first node v of T such that $label(v) = label(u) = e$ is node 5. As above, we have an embedding, and the value of $v = 5$ is written into $e(2, q)$ for the left relatives $q = 0, \ldots, 4$ of v. Again the remaining nodes $v = 6, \ldots, 9$ are scanned without encountering any matching labels.

Finally, $u = 3 = root(P)$. Node $v = 7$ is the first node of T with $label(v) = label(u) = a$. The node p preceding $desc(v)$ in postorder is node 4. The first child of u is node number 1. Its image in $rr(p) \supset desc(v)$ is looked up from $e(1, 4)$; value $n + 1 = 10$ means that there is no embedding of the child in $desc(v)$. Next, a similar failure occurs with $v = 8$ and $p = 2$. Finally $v = 9 = root(T)$, $label(v) = label(u) = a$, which leads to testing the embedding of the subtrees of u by executing $p := 0$, $p := e(1, 0) = 1$, and $p := e(2, 1) = 5$. The algorithm has found a root-preserving embedding of P in T. The value of $v = 9$ is written as the final result into $e(3, 0)$. Since 0 is the only left relative of 9, the computation ends.

THEOREM 6.2. *Algorithm* 6.1 *fills table e correctly.*

Proof. Consider a fixed pattern node u. We outline a proof that all columns of row u get correct values in the **while**-loop on lines 15–18 of the algorithm. First, the precondition that all columns up to q have got the right values is initially true when $q = 0$. We show that the following invariant holds on line 16:

$$\exists f \in R(P[u], P[v]) \quad \wedge \quad v \in rr(q) \tag{2}$$

$$\wedge\, \forall 1 \leq x < v : (\exists g \in R(P[u], T[x]) \quad \Rightarrow \quad x \notin rr(q)) \,. \tag{3}$$

The invariant shows that the loop writes correct values into $e(u, q)$. Since q increases only in the loop, the truth of the invariant maintains the precondition for the subsequent executions of the loop.

Assume that u is a leaf; then $e(u, q)$ should be assigned the number of the first node v in $rr(q)$ such that $label(v) = label(u)$. It is clear that (2) holds when we are on line 16. (Note that $q \in lr(v)$ and $v \in rr(q)$ are equivalent.) When line 16 is executed for the first time, (3) is vacuously true. By Lemma 4.6 we can strengthen (3) into

$$\forall\, 1 \le x < v, \forall\, y \ge q : (\exists\, g \in R(P[u], T[x]) \Rightarrow x \notin rr(y)) \,. \tag{4}$$

At the exit from the loop, $v \notin rr(q)$ allows us to deduce from (4) that

$$\forall\, 1 \le x \le v, \forall\, y \ge q : (\exists\, g \in R(P[u], T[x]) \Rightarrow x \notin rr(y)) \,. \tag{5}$$

This postcondition makes (3) true on the subsequent executions of the loop. It also shows that the writing is complete, i.e., value v must not be written into $e(u, y)$ for any column $y \ge q$.

Next, assume that u is a nonleaf node, and the rows of the table e corresponding to the children of u have been correctly computed. Then, as in Algorithm 5.1, the **while**-loop on lines 10–13 finds a left embedding of the subtrees of u in the subtrees of v, if there is any. The correctness of the **while**-loop on lines 13–16 is verified as in the base case. □

THEOREM 6.3. *Algorithm* 6.1 *requires* $O(mn)$ *time and space.*

Proof. Space: Table e requires $O(mn)$ space.

Time: During every execution of the outermost loop q may increase in steps of one from 0 to n. Therefore, the **while**-loop that increments q requires $O(n)$ steps per one outermost loop. One execution of the **while**-loop on lines 10–13 requires time $O(1 + |u|)$, where $|u|$ is the out-degree of node u. We get total time

$$O\left(\sum_{u=1}^{m}(n + \sum_{v=1}^{n}(1 + |u|))\right) = O\left(n \sum_{u=1}^{m}(2 + |u|)\right) \,.$$

The sum $\sum_{u=1}^{m} |u|$ equals the number of edges in tree P, which is $m - 1$. Therefore the total time is $O(n(3m - 1)) = O(mn)$. □

7. Unordered tree inclusion. So far we have presented an efficient algorithm for the tree inclusion problem in the case of ordered trees. Now we turn to the unordered version of the problem, which appears to be essentially harder.

A tree P is an *unordered included tree* of a tree T if the nodes of P can be injectively mapped onto the nodes of T preserving the labels and the ancestorship relation between nodes. Such a mapping is called an *unordered embedding*. We do not require the left-to-right order of the nodes to be preserved in an unordered embedding.

As before, the tree inclusion can be characterized also operationally.

LEMMA 7.1. *A tree P is an unordered included tree of a tree T if and only if P can be obtained from T by deleting nodes and permuting subtrees.*

The existence of an efficient algorithm for unordered tree inclusion is unlikely, since the problem appears to be NP-complete. We prove the NP-completeness of unordered tree inclusion by a reduction from the basic NP-complete problem SATISFIABILITY [3], [5]. For proving the NP-completeness of unordered tree inclusion we use the following lemma stating that a slight restriction of SATISFIABILITY is still NP-complete.

LEMMA 7.2. *Let $U = \{u_1, \ldots, u_n\}$ be a set of Boolean variables and $C = \{c_1, \ldots, c_m\}$ be a collection of clauses over U. Now C can be transformed in polynomial time into such a collection of clauses $C' = \{c'_1, \ldots, c'_{m'}\}$ that*

1. *C is satisfiable if and only if C' is satisfiable, and*
2. *no negated variable occurs in two clauses of C'.*

Proof. Let $D = \{\{u, y_u\}, \{\bar{u}, \bar{y}_u\} \mid u \in U\}$, where y_u is a new variable, for each $u \in U$. Clauses $\{u, y_u\}$ and $\{\bar{u}, \bar{y}_u\}$ express exclusive or of u and y_u, i.e., a truth assignment can satisfy

them only if it assigns opposite values to u and y_u. Let E be the set of clauses obtained from C by replacing each negated occurrence $\bar{u}$ of a variable by y_u, and let $C' = D \cup E$. Now C' is satisfiable if and only if C is satisfiable. The transformation obviously can be done in polynomial time. □

THEOREM 7.3. *The unordered tree inclusion problem is NP-complete.*

Proof. It is easy to see that the problem is in NP: An algorithm can guess the mapping of the pattern nodes onto the target nodes and check in polynomial time that it is indeed an embedding.

The completeness for NP is shown by a reduction from SATISFIABILITY. Let an instance of SATISFIABILITY be given by a collection of variables $U = \{u_1, \ldots, u_n\}$ and a collection of clauses $C = \{c_1, \ldots, c_m\}$ over U. By Lemma 7.2 we can assume that no negated variable appears in two clauses of C.

Form a pattern tree P and a target tree T of unordered tree inclusion problem as follows. Let $P = (N_P, E_P)$ be a tree given by nodes $N_P = \{0, \ldots, m\}$ and by parent-child edges

$$E_P = \{(0, x) \mid x \in N_P, x \neq 0\} .$$

Let $label(x) = x$ for all $x \in N_P$. The intuition is that the nodes of tree P, excluding the root, represent clauses of C and each of them is labeled by the index of the corresponding clause. Let $T = (N_T, E_T)$ be the tree whose nodes consist of pairs

$$\begin{aligned} N_T \quad = \quad & \{(0, 0)\} \cup \\ & \{(u, j) \mid u \in c_j \in C\} \cup \\ & \{(\bar{u}, j) \mid \bar{u} \in c_j \in C\} , \end{aligned}$$

and whose parent-child edges are

$$\begin{aligned} E_T \quad = \quad & \{((0, 0), (u, j)) \mid u \in c_j \in C, \bar{u} \notin \cup C\} \cup \\ & \{((0, 0), (\bar{u}, j)) \mid \bar{u} \in c_j \in C\} \cup \\ & \{((\bar{u}, j), (u, k)) \mid \bar{u} \in c_j \in C, u \in c_k \in C\} . \end{aligned}$$

So, tree T has one node corresponding to each occurrence of a literal in a clause of C, and an additional root node $(0, 0)$. The nodes for the positive occurrences of variables that do not occur negated in C are children of the root of T. A node $(\bar{u}, j)$ corresponding to a negated occurrence of variable u is a child of the root of T and the parent of the nodes corresponding to the positive occurrences of u. The assumption of unique occurrences of negative literals implies that each node except the root has a unique parent, and thus T is indeed a tree. Let $label((x, j)) = j$ for all nodes $(x, j) \in N_T$, i.e., the nodes corresponding to the literal occurrences in a clause c_j are labeled by j. An example of the construction is shown in Fig. 6.

Obviously, the trees can be formed in polynomial time.

Now we claim that there is a satisfying truth assignment for C if and only if P is an unordered included tree of T. First, assume that t is a satisfying truth assignment for C. Define a mapping h from the nodes of P onto the nodes of T as follows: For the root of P set $h(0) = (0, 0)$ and for other nodes j of P set $h(j) = (l, j)$, where $(l, j) \in N_T$ is some node such that literal l is true under truth assignment t; such a node can be selected because t satisfies at least one literal of every clause $c_j \in C$. Now h is an unordered embedding, since it is obviously injective and label preserving, and it cannot map two sibling nodes from P to an ancestor and its descendant in T. Otherwise, by the construction of T, truth assignment t would satisfy both a positive and a negative occurrence of the same variable, which is impossible.

Next assume that h is an unordered embedding of P in T. Set $t(x) =$ false if the range of h contains a node $(\bar{x}, j)$ corresponding to a negative occurrence of variable x, and $t(x) =$ true

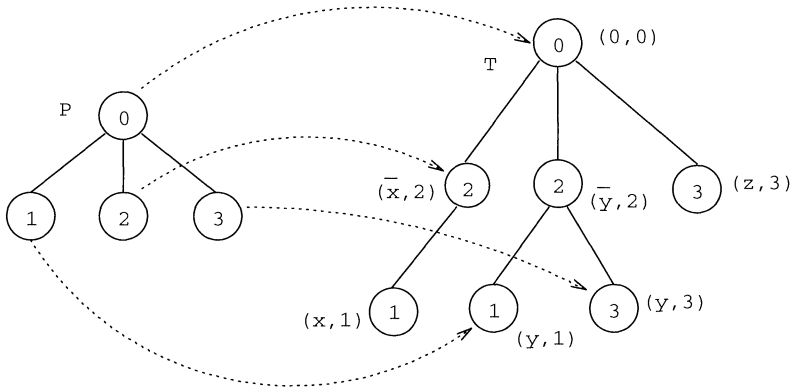


FIG. 6. *Trees for clauses* $c_1 = \{x, y\}$, $c_2 = \{\bar{x}, \bar{y}\}$, *and* $c_3 = \{y, z\}$. *The embedding shown by arrows corresponds to satisfying truth assignments that set* x *to false and* y *to true.*

if the range of h contains a node (x, j) corresponding to a positive occurrence of variable x. It is easy to see that t is a well-defined truth assignment for a subset of variables in C, and that t satisfies at least one literal in each clause of C. □

8. An algorithm for unordered tree inclusion. In this section we present an algorithm for solving the unordered tree-inclusion problem. The algorithm requires exponential time in the worst case. In the light of the NP-completeness of the problem, it is unlikely that we can do any better. We will see that the difficulty of solving the problem lies in permuting the left-to-right order of the pattern nodes. If the size or the branching of the pattern is limited, the problem becomes feasible to our algorithm.

Let P be the pattern tree and T the target tree. The next algorithm is based on manipulating match systems consisting of subsets of pattern nodes. The *match system* $S(v)$ for a target node v consists of all the subsets $\{u_1, \ldots, u_k\}$ of pattern nodes such that the forest $\langle P[u_1], \ldots, P[u_k]\rangle$ has an unordered embedding in $T[v]$. The algorithm computes the match systems for each target node while going through the target in a bottom-up order. Note that if v' is an ancestor of a target node v, then $T[v] \sqsubseteq T[v']$ and therefore $S(v) \subseteq S(v')$. Also, for each pattern node u we have that $\{u\} \in S(v)$ only if $children(u) \in S(v)$.

ALGORITHM 8.1. *Unordered tree inclusion algorithm.*
Input: *Trees* P *and* T ($|P| = m$, $|T| = n$).
Output: *The nodes* v *of* T *such that there is a root-preserving unordered embedding of* P *in* $T[v]$.
Method:

```
1.    for v := 1, ..., n do
2.    comment: Go through the target nodes in postorder;
3.        S := {∅};
4.        Let v_1, ..., v_l, l ≥ 0, be the children of v;
5.        for i := 1, ..., l do
6.            S := {A ∪ B | A ∈ S, B ∈ S(v_i)};
7.        od;
8.        SΔ := ∅;
9.        for all u ∈ P such that label(u) = label(v) do
10.           if children(u) ∈ S then
11.               SΔ := SΔ ∪ {{u}};
```

```
12.              fi;
13.          od;
14.          if {root(P)} ∈ SΔ then
15.          comment: An occurrence found;
16.              output v;
17.          fi;
18.          S(v) := S ∪ SΔ;
19.      od.
```

The algorithm computes the match system $S(v)$ for a target node v with children $v_1, \ldots, v_l$ as follows. First the loop on lines 5–7 computes from the match systems of the children of v a set S. The invariant for the loop says that the set S consists of the subsets $\{u_1, \ldots, u_k\}$ of such pattern nodes that the forest $\langle P[u_1], \ldots, P[u_k]\rangle$ has an unordered embedding in the forest $\langle T[v_1], \ldots, T[v_i]\rangle$. For $i = 0$ this is clearly true, since $S = \{\emptyset\}$. Next, assume inductively that the invariant holds for $i - 1$ when $0 < i \leq l$, and that $S(v_i)$ is the correct match system for the child v_i of target node v. Now it is rather easy to see that the forest $\langle P[u_1], \ldots, P[u_k]\rangle$ has an unordered embedding in $\langle T[v_1], \ldots, T[v_i]\rangle$ if and only if $\{u_1, \ldots, u_k\} = A \cup B$ for some $A \in S$ and $B \in S(v_i)$.

Next the algorithm computes in a set $S\Delta$ the singleton sets $\{u\}$ of the pattern nodes u for which there is a root-preserving embedding of $P[u]$ in $T[v]$; these are exactly those nodes u whose label matches the label of v and whose set of children belongs to S. After this process, $S \cup S\Delta$ is the the match system of v.

We will restrict match systems to consist of sets of sibling nodes only. (Note that they only affect the insertion of new pattern nodes to the match systems on line 10 of Algorithm 8.1.) This can be done by restricting, on line 6, to uniting only sets A and B of nodes that have a common parent.

The size of the match systems depends on P only. If P is fixed, executing line 6 and the loop on lines 9–13 takes constant time. The algorithm examines every target node at most twice; once as v and at most once as a child of v. Therefore we have the following result.

THEOREM 8.2. *For a fixed pattern P, an instance (P, T) of the unordered tree inclusion problem can be solved in time $O(|T|)$.*

If we are to solve the problem repeatedly with the same pattern P, we can solve it in two phases. The first phase is to perform a preprocessing on P, which may take a long time as a function of $|P|$. After the preprocessing a modified version of Algorithm 8.1 needs time that is only linear in $|T|$ and not a function of $|P|$. The preprocessing would be based on enumerating the collection $\mathcal{S}(P)$ of possible values of variable S, and using this enumeration to index arrays storing the possible assignments performed on lines 6, 9–13 and 18. A similar approach can be found in the bottom-up tree pattern matching algorithm of [8].

The preprocessing is reasonable only when the size of $\mathcal{S}(P)$ is not too large. The size of collection $\mathcal{S}(P)$ depends strongly on the form of the pattern P. If P consists of m nodes that form a path from the root to a single leaf, $\mathcal{S}(P)$ consists of $m + 1$ systems that are $\{\emptyset\}$, $\{\emptyset, \{1\}\}, \ldots,$ and $\{\emptyset, \{1\}, \ldots, \{m\}\}$. At the other extreme, the size of $\mathcal{S}(P)$ can be doubly exponential in m. To see this, consider a pattern P consisting of a root node and $m - 1$ leaves. Let $\mathcal{A}$ be the system consisting of the sets of $\lfloor (m-1)/2 \rfloor$ leaves of P. Obviously $|\mathcal{A}| = \binom{m-1}{\lfloor (m-1)/2 \rfloor}$, and every subset of $\mathcal{A}$ belongs to $\mathcal{S}(P)$. Therefore

$$|\mathcal{S}(P)| > 2^{\binom{m-1}{\lfloor (m-1)/2 \rfloor}}.$$

Each match system S is *monotone decreasing*, i.e., if $A \in S$, then $B \in S$ for each $B \subseteq A$. Thus we can represent the match systems as *Sperner systems* by maintaining their maximal

elements only. The following results give an upper bound for the number of Sperner systems on m nodes.

PROPOSITION 8.3 [23], [1]. *The size of a Sperner system on an m-element set is at most* $\binom{m}{\lfloor m/2 \rfloor}$.

PROPOSITION 8.4 [2]. *The number of set systems $\mathcal{A}$ on an m-element set, satisfying*

$$|\mathcal{A}| \leq c\binom{m}{\lfloor m/2 \rfloor}, \tag{6}$$

is at most

$$2^{\frac{c}{2}\binom{m}{\lfloor m/2 \rfloor}\log m(1+o(1))}.$$

Putting together these propositions gives the following upper bound for the number of Sperner systems on an m-element set, denoted by $\alpha(m)$:

$$\alpha(m) \leq 2^{\frac{1}{2}\binom{m}{\lfloor m/2 \rfloor}\log m(1+o(1))}$$

When the systems in $\mathcal{S}(P)$ are Sperner systems this is also an upper bound for the size of $\mathcal{S}(P)$. As discussed above, a pattern consisting of a root node and $m-1$ leaves brings $\mathcal{S}(P)$ fairly close to this bound.

Let us next analyze how Algorithm 8.1 performs with patterns whose branching is limited. The executions of line 6 will dominate the execution time of the algorithm. Let k be the largest out-degree of any node in P. Any set of sibling nodes of P has at most 2^k subsets. On line 6 we need to unite only subsets A and B that consist of children of a common parent at a time. For each of the $|P|$ parents in the pattern there will be at most $O(2^k 2^k)$ pairs of such sets. Each union $A \cup B$ can be formed in $O(k)$ time and the union can be checked for not being a duplicate of any other member of the resulting system S in $O(k2^k)$ time.

The rest of the algorithm can be executed in $O(|P|)$ time per one target node by choosing a reasonable data structure for S and $S\Delta$. We have derived the following upper bound for the time complexity of the unordered tree-inclusion problem.

THEOREM 8.5. *If the out-degrees of the nodes of P are bounded by k, the unordered tree-inclusion problem is solvable in time $O(|P|k2^{2k}|T|)$. Specifically, if k is a constant, the problem is solvable in $O(|P||T|)$ time, and if k is $O(\log|T|)$, the problem is solvable in time $O(|P|\log|T||T|^3)$.*

A similar but less precise result could also be obtained by using the dynamic programming approach.

9. Conclusions. We have considered the tree-inclusion problem, which arises from database query processing. We have given a dynamic processing solution requiring $O(mn)$ time, where m and n are the sizes of the trees. The algorithm is faster than the previous ones. The corresponding problem for unordered trees is considerably more difficult. We have given a proof of its NP-completeness. For the unordered problem we have presented an algorithm that works in $O(mn)$ time if the out-degrees of the pattern nodes are bounded by a constant, and in polynomial time if they are $O(\log n)$.

There are several open problems. One is improving the running time of the algorithm for the ordered problem. Breaking the mn-barrier seems rather hard, however. Another promising area is trying to reduce some matching problems to the tree-inclusion problem; this could give upper or lower bounds for the complexity of this problem.

In an application where the target tree is very large, the $\Theta(mn)$ space and time requirements of Algorithm 6.1 may be unacceptable. Another algorithm that solves the ordered tree-inclusion problem in $O(m\,depth(T))$ space has been presented in [10]. The idea of this

algorithm is similar to Algorithm 8.1, with the difference that in the ordered case the match systems have a more economical representation. The complexity of the new algorithm is sensitive to the instances of the problem: it runs in time $O(c_P(T)\,n)$, where $c_P(T)$ is the number of the subtrees of P that are included in T. If no part of the pattern appears in the target, the algorithm runs in time $O(n)$.

Acknowledgments. We wish to thank Kari-Jouko Räihä and an anonymous referee for useful comments on previous versions of the article.

REFERENCES

[1] B. Bollobás, *Combinatorics*, Cambridge University Press, Cambridge, UK, 1986.

[2] G. Burosch, J. Demetrovics, G. O. H. Katona, D. J. Kleitman, and A. A. Sapozhenko, *On the number of databases and closure operations*, Theoret. Comput. Sci., 78 (1991), pp. 377–381.

[3] S. A. Cook, *The complexity of theorem-proving procedures*, in Proc. of the 3rd Annual ACM Symposium on Theory of Computing, ACM Press, New York, 1971, pp. 151–158.

[4] M. Dubiner, Z. Galil, and E. Magen, *Faster tree pattern matching*, in Proc. of the Symposium on Foundations of Computer Science (FOCS'90), St. Louis, Missouri, IEEE Computer Society Press, 1990, pp. 145–150.

[5] M. R. Garey and D. S. Johnson, *Computers and Intractability*, W. H. Freeman Co., San Francisco, CA, 1979.

[6] G. H. Gonnet and F. Wm. Tompa, *Mind your grammar - a new approach to text databases*, in Proc. of the Conference on Very Large Data Bases (VLDB'87), Brighton, Morgan Haufman Publishers, 1987, pp. 339–346.

[7] D. S. Hirschberg, *A linear space algorithm for computing maximal common subsequences*, Comm. of the ACM, 18 (1975), pp. 341–343.

[8] C. M. Hoffman and M. J. O'Donnell, *Pattern matching in trees*, J. Assoc. Comput. Mach., 29 (1982), pp. 68–95.

[9] D. S. Johnson, *The NP-completeness column: An ongoing guide*, J. Algorithms, 8 (1987), pp. 285–303.

[10] P. Kilpeläinen, *Tree Matching Problems with Applications to Structured Text Databases*, Ph.D. thesis, University of Helsinki, Dept. of Comp. Science, November 1992.

[11] P. Kilpeläinen, G. Lindén, H. Mannila, and E. Nikunen, *A structured document database system*, in R. Furuta, ed., *EP90 – Proceedings of the International Conference on Electronic Publishing, Document Manipulation & Typography*, the Cambridge Series on Electronic Publishing, Cambridge University Press, Cambridge, UK, 1990.

[12] P. Kilpeläinen and H. Mannila, *The tree inclusion problem*, in S. Abramsky and T. S. E. Maibaum, eds., *TAPSOFT'91, Proc. of the International Joint Conference on the Theory and Practice of Software Development, Vol. 1: Colloqium on Trees in Algebra and Programming (CAAP'91)*, Springer-Verlag, New York, 1991, pp. 202–214.

[13] ———, *Retrieval from hierarchical texts by partial patterns*, in R. Korfhage, E. Rasmussen, and P. Willett, ed., *SIGIR '93 – Proceedings of the Sixteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 214 – 222, ACM Press, New York, 1993.

[14] ———, *Query primitives for tree-structured data*, in M. Crochemore and D. Gusfield, ed., Proceedings of the Fifth Annual Symposium on Combinatorial Pattern Matching, Tuscon, Arizona, pp. 213–225, Springer-Verlag, 1994.

[15] D. E. Knuth, *The Art of Computer Programming*, Vol. 1, Addison-Wesley, Reading, MA, 1969.

[16] S. R. Kosaraju, *Efficient tree pattern matching*, in Proc. of the Symposium on Foundations of Computer Science (FOCS'89), Research Triangle Park, NC, IEEE Computer Society Press, 1989, pp. 178–183.

[17] H. Mannila and K.-J. Räihä, *On query languages for the p-string data model*, in H. Kangassalo, S. Ohsuga, and H. Jaakkola, ed., *Information Modelling and Knowledge Bases*, IOS Press, Amsterdam, 1990, pp. 469–482.

[18] J. Matoušek and R. Thomas, *On the complexity of finding iso- and other morphisms for partial k-trees*, Discrete Mathematics ,108 (1992), pp. 343–364.

[19] R. Ramesh and I. V. Ramakrishnan, *Nonlinear pattern matching in trees*, J. Assoc. Comput. Mach., 39 (1992), pp. 295–316.

[20] S. W. Reyner, *An analysis of a good algorithm for the subtree problem*, SIAM J. Comput., 6 (1977), pp. 730–732.

[21] N. Robertson and P. D. Seymour, *Graph minors. II. Algorithmic aspects of tree-width*, J. Algorithms, 7 (1986), pp. 309–322.

[22] D. SHASHA AND K. ZHANG, *Fast algorithms for the unit cost editing distance between trees*, J. Algorithms, 11 (1990), pp. 581–621.

[23] E. SPERNER, *Ein Satz über Untermengen einer endlichen Menge*, *Math. Z.*, 27 (1928), pp. 544–548.

[24] J. TAGUE, A. SALMINEN, AND C. MCCLELLAN, *Complete formal model for information retrieval systems*, in A. Bookstein, Y. Chiaramella, G. Salton, and V. V. Raghavan, eds., *Proceedings of the Fourteenth Annual International ACM/SIGIR Conference on Research and Development in Information Retrieval*, ACM Press, New York, 1991, pp. 14–20.

[25] K.-C. TAI, *The tree-to-tree correction problem*, J. Assoc. Comput. Mach., 26 (1979), pp. 422–433.

[26] R. M. VERMA AND S. W. REYNER, *An analysis of a good algorithm for the subtree problem, corrected*, SIAM J. Comput., 18 (1989), pp. 906–908.

[27] K. ZHANG AND D. SHASHA, *Simple fast algorithms for the editing distance between trees and related problems*, SIAM J. Comput., 18 (1989), pp. 1245–1262.

SIAM J. COMPUT.
Vol. 24, No. 2, pp. 357–368, April 1995

011

RANDOMIZED INTERPOLATION AND APPROXIMATION OF SPARSE POLYNOMIALS*

YISHAY MANSOUR†

Abstract. We present a randomized algorithm that interpolates a sparse polynomial in polynomial time in the bit complexity model. The algorithm can be also applied to approximate polynomials that can be approximated by sparse polynomials (the approximation is in the L_2 norm).

Key words. polynomial interpolation, randomized algorithms, approximation, harmonic analysis

AMS subject classifications. 65D05, 65T99, 41A10

1. Introduction.

We consider the problem of interpolating a sparse polynomial with integer coefficients. The interpolation model is the following. The algorithm can query the polynomial P on any input x, and receives the value $P(x)$ with some finite accuracy. The accuracy parameter determines the number of significant bits that are given from the value of $P(x)$, which helps truncate $P(x)$ after that number of significant bits. The running time of the algorithm is the number of bit operations performed.

The parameters of the sparse polynomial are n—the number of variables, t—an upper bound on the number of nonzero coefficients, d—an upper bound on the degree of a single variable, and L—an upper bound on the absolute value of the largest coefficient. A t-sparse polynomial is a polynomial with at most t nonzero coefficients. The number of bits required to express a t-sparse polynomial is $O(t(\log L + n \log d)) = \text{poly}(n, t, \log d, \log L)$.[1]

Our main result is a randomized algorithm that interpolates a t-sparse polynomial in time $\text{poly}(n, t, \log d, \log L)$. In the analysis of the algorithm, we count the number of operations performed, but since the accuracy that our algorithm requires is very small (logarithmic in the above parameters), the running time remains $\text{poly}(n, t, \log d, \log L)$ in the bit complexity model.

An interesting application of our algorithm is finding an *approximation* by a sparse polynomial. The approximation is in the L_2 norm, i.e., the expected error squared is small. The main result is that if a polynomial P can be approximated by a t-sparse polynomial Q such that the power of $P - Q$ is bounded by ϵ, then our algorithm can find a t-sparse polynomial Q' such that the power of $P - Q'$ is bounded by $O(\epsilon)$. (The *power* of a polynomial is the sum of the squares of its coefficients.) Our algorithm achieves this by finding only the "large" coefficients of P, rather than finding all of the coefficients of P. This notion of approximation is especially interesting when the values of the variables have approximately unit magnitude. We believe that this notion of approximation is especially interesting in our framework, since if we allow the value of the variable to be, for example, 2, then the output of the polynomial may have already d bits.

Interpolation of a univariate polynomial can be performed in time $O(d \log d)$, using the Fast Fourier Transform (FFT). The FFT does not use the information about the sparsity of the polynomial, therefore, its running time is polynomial in d, rather than t and $\log d$. Since the FFT algorithm does not consider the sparsity of the polynomial, for multivariate polynomials where the number of possible coefficients is exponential, its running time is exponential.

*Received by the editors October 13, 1992; accepted for publication (in revised form)October 28, 1993.

†Dept. of Computer Science, Tel-Aviv University, Tel-Aviv, Israel and IBM - T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598. Part of this research was completed while the author was at Aiken Computation Laboratory, Harvard University, Cambridge, Massachusetts.

[1] We denote by poly(a) a function which is bounded by a polynomial in a, i.e., $a^{O(1)}$.

Previous works on sparse multivariate polynomial interpolation were mainly in the algebraic model, where each operation requires one unit of time. When considering those works in the bit complexity model, one has to take into account the number of bits which were given to the polynomial, and also the size of the result of the polynomial and the required accuracy on the result. Zippel [Zip79] gave a probabilistic algorithm that runs in time $\text{poly}(n, t, d)$. His algorithm evaluates the polynomial at points of size $\Theta(\log ntd)$ bits and requires $\Theta(d \log ntd + \log L)$ bits, which implies that the running time in the bit complexity model is $\text{poly}(n, t, d, \log L)$.

Grigoriev and Karpinski [GK87] showed a deterministic NC algorithm to compute a matching, in the case that the permanent is sparse. (The permanent can be viewed as a generic multivariate polynomial.) Ben-Or and Tiwari [BOT88] gave a deterministic algorithm to compute a sparse multivariate polynomial that runs in time $\text{poly}(n, t, \log d)$. The algorithm evaluates the polynomial at points of size $O(t \log n)$ bits and requires accuracy of $O(td \log n + \log L)$ bits. Therefore, the running time of their algorithm in the bit complexity model requires time $\text{poly}(n, t, d, \log L)$. Grigoriev, Karpinski, and Singer [GKS90b] showed how to interpolate a rational function with $O(nt^t)$ points, which is independent of the degree d ! However, in the bit complexity model the running time is $\text{poly}(n, t^t, d, \log L)$. Algorithms for interpolation of sparse polynomials over finite field appears in [GKS90a] and [Kar89].

The work here extends the technique developed in [GL89] and [KM91] for the Fourier transform over the hypercube. The extension handles a ring of integers modulo an integer (i.e., Z_d). After we show how to handle Z_d, we use the previous techniques and extend the result to handle Z_d^n.

Remark. In this work, we are interested in the bit complexity model. In the analysis, we use the algebraic model and count the number of operations. By guaranteeing that the precision that our algorithm requires is small, we make sure that its running time in both models is similar.

The paper is organized as follows. Section 2 mentions a few known facts about the Chernoff bounds, discrete Fourier transform, and defines the black box model. Section 3 has the interpolation to univariate sparse polynomials with small coefficients, which is extended in §4 to handle arbitrary coefficients. The precision accuracy that the algorithm requires is discussed in §5. Section 6 deals with the approximation of a real valued function by sparse polynomials. Section 7 extends the case of univariate polynomials to multivariate polynomials.

2. Preliminaries. Chernoff bounds. In the analysis of our randomized algorithms, we use Chernoff bounds. The following theorem formalizes the bounds in a way which would be most useful for us. (See [HR89].)

THEOREM 2.1 (Chernoff bound). *Let $X_1, \ldots, X_m$ be i.i.d with $E[X_i] = p$, $|X_i| \leq H$ and let $S_m = X_1 + \cdots + X_m$. If $m \geq \frac{2H^2}{\epsilon^2} \ln \frac{2}{\delta}$, then*

$$Pr\left[\left|\frac{S_m}{m} - p\right| > \epsilon\right] \leq \delta.$$

Complex numbers. The magnitude of a complex number $z = a + ib$ is $\|z\| = \sqrt{a^2 + b^2}$. Let $\omega_d = e^{\frac{2\pi}{d} i}$ be a dth root of unity. Recall that $\|\omega_d^k\| = 1$ for any d and k.

Discrete Fourier transform. We list here a few known facts about the discrete Fourier transform. Consider a polynomial $P(x) = \sum_{j=0}^{d-1} a_j x^j$. Each coefficient of the polynomial satisfies

$$a_j = \frac{1}{d} \sum_{k=0}^{d-1} P(\omega_d^k) \omega_d^{-kj}.$$

Parseval identity relates the sum of the squares of the coefficients to the values of the polynomial as follows:

$$\sum_{j=0}^{d-1} a_j^2 = \frac{1}{d}\sum_{k=0}^{d-1} P(\omega_d^k)P(\omega_d^{-k}) = \frac{1}{d}\sum_{k=0}^{d-1} \|P(\omega_d^k)\|^2.$$

For any integer k not divisible by d,

$$\sum_{m=0}^{d-1} (\omega_d^k)^m = 0.$$

The black box model. Our algorithm assumes that it has a black box that evaluates the polynomial P. The algorithm evaluates the polynomial at roots of unity of the form ω_d^k. The interaction between the algorithm and the black box is the following.

To evaluate the polynomial P at ω_d^k, the algorithm gives the black box the pair (d, k). Denote by $\rho(k, d)$ the output of the black box on input (d, k).

The black box has *precision accuracy* ϵ if for any input (d, k), the absolute error is at most ϵ, i.e.,

$$\|P(\omega_d^k) - \rho(d, k)\| \leq \epsilon.$$

Let $H = \max_k\{\|P(\omega_d^k)\|\}$. The number of bits required for precision accuracy ϵ is $O(\log H + \log \frac{1}{\epsilon})$.

3. Univariate polynomials. In this section we discuss interpolation of t-sparse polynomials with integer coefficients. The result is based on a searching technique that we develop to search in the space of the possible coefficients. The search technique is similar to the one in [GL89] and [KM91].

Given a polynomial $P(x)$, we split it into two polynomials: one that includes the coefficients of odd degrees (denoted by $P_{1,1}$), and the other which includes the coefficients of even degrees (denoted by $P_{0,1}$). Note that $P_{0,1}(x) = (P(x) + P(-x))/2$ and $P_{1,1}(x) = (P(x) - P(-x))/2$. In a similar way we define $P_{\alpha,\ell}(x)$, where $0 \leq \alpha \leq 2^\ell - 1$ and $0 \leq \ell \leq \log d$, as follows:

$$P_{\alpha,\ell}(x) = \sum_{j:\, e_j \bmod 2^\ell = \alpha} a_{e_j} x^{e_j}.$$

The algorithm works as follows. It keeps a list of the current candidates polynomials, $P_{\alpha,\ell}$. The algorithm takes a polynomial $P_{\alpha,\ell}$ off the list and tests if it is identically zero. If the polynomial is identically zero no action is taken. (This implies that we terminate the search for coefficients of exponents e, $e = \alpha \bmod 2^\ell$, since all those coefficients are zero.) Otherwise, we continue recursively searching, i.e., we add to the list of candidates polynomials the polynomials $P_{\alpha+2^\ell,\ell+1}$ and $P_{\alpha,\ell+1}$. When $\ell = \log d$, then we have reached a nonzero coefficient. (Namely, the polynomial $P_{\alpha,\log d}(x) = a_\alpha x^\alpha$.)

Since there are at most t nonzero coefficients and for each coefficient we perform $\log d$ recursive calls, the total number of recursive calls is bounded by $O(t \log d)$.

We are not able to compute the values of the polynomials that we create *exactly*, but rather we are able to approximate them very well. One way to overcome this is to approximate the value of a polynomial at randomly selected points, and if at some point it is more than $1/2$ then we assume that it is a nonzero polynomial. Here we choose an alternative way to test for zero polynomials that is based on approximating the power of the polynomial. Since the polynomial has integer coefficients, if it is a nonzero polynomial then it has power at least one. We approximate the power of the polynomial and if the approximation is more than $1/2$, we assume that the polynomial is nonzero. We prefer the method that is based on approximating the power since it extends more naturally to the case of large coefficients and to the case of approximation by sparse polynomials.

3.1. The main procedure. In this subsection, we outline the main idea of how to recover the coefficients of a t-sparse polynomial $P(x) = \sum_{j=1}^{t} a_{e_j} x^{e_j}$, whose coefficients are integers. The basic strategy is to recover the exponents e_j bit after bit, starting from the least significant bit.

Allow d to be the smallest power of two greater than $\max_j\{e_j\}$, i.e., $d = 2^{\lceil \log(1+\max_j\{e_j\})\rceil}$. Consider the sets $S_\ell = \{e_j \bmod 2^\ell : 1 \le j \le t\}$ for $0 \le \ell \le \log d$. The set S_ℓ includes the ℓ least significant bits of the exponents e_j. Clearly, the size of a set S_ℓ is bounded by t, and the set S_0 includes only zero. We show how to compute the set $S_{\ell+1}$ given the set S_ℓ.

From the definition of S_ℓ it follows that if $\alpha \notin S_\ell$ then $P_{\alpha,\ell}(x) \equiv 0$, while if $\alpha \in S_\ell$ then $P_{\alpha,\ell}(x) \not\equiv 0$. One step in the construction is testing if a polynomial is identically zero. Assume that we are given such a procedure $ZERO(P_{\alpha,\ell})$ that returns $TRUE$ if and only if $P_{\alpha,\ell} \equiv 0$ (later we will show how to implement it).

Given the set S_ℓ we construct the set $S_{\ell+1}$ as follows. For each $\alpha \in S_\ell$ we create two integers: $\alpha_0 = \alpha$ and $\alpha_1 = \alpha + 2^\ell$. For each α_σ, $\sigma \in \{0, 1\}$, we test $ZERO(P_{\alpha_\sigma,\ell})$. If $ZERO(P_{\alpha_\sigma,\ell})$ is $FALSE$ we add α_σ to $S_{\ell+1}$. Since the set $S_{\log d}$ includes all the exponents of P, when we recover $S_{\log d}$ we have found all the exponents of P.

CLAIM 3.1. *The algorithm constructs each S_ℓ correctly.*

Proof. The proof is by induction on ℓ. Initially S_0 includes only zero, so it is correct. Assume that we constructed S_ℓ correctly. We have to show two properties of $S_{\ell+1}$: that any element added should have been added and that every element of $S_{\ell+1}$ was added at some point.

An element $\beta = \alpha_\sigma$, $\sigma \in \{0, 1\}$, is added to $S_{\ell+1}$ only if $ZERO(P_{\beta,\ell+1})$ is $FALSE$. This implies that $\sum_{j:e_j \bmod 2^{\ell+1}=\beta} a_{e_j}^2 \neq 0$. Therefore, there exists some coefficient a_{e_j} such that $e_j \bmod 2^{\ell+1} = \beta$, which is not zero.

For the second part assume that $\beta \in S_{\ell+1}$. Let $\beta \bmod 2^\ell = \alpha$. By the induction hypothesis $\alpha \in S_\ell$. When we handle α, in one of the two cases we test $\alpha_\sigma = \beta$. Since $\beta \in S_{\ell+1}$, $ZERO(P_{\beta,\ell+1}) = ZERO(P_{\alpha_\sigma,\ell+1})$ is $FALSE$. Hence β was added to $S_{\ell+1}$. □

LEMMA 3.2. *The number of calls to the procedure $ZERO$ is bounded by $O(t \log d)$.*

Proof. There are $\log d$ sets S_ℓ. The size of each set S_ℓ is bounded by t. For each element $\alpha \in S_\ell$ we perform two calls, $ZERO(\alpha_0, \ell)$ and $ZERO(\alpha_1, \ell)$. □

So far we have an algorithm that requires an oracle for computing $P_{\alpha,\ell}(x)$ and the procedure $ZERO$. In the next subsection we show how to approximate $P_{\alpha,\ell}$, and in the subsection after that we show how to approximate $ZERO$.

3.2. Approximating $P_{\alpha,\ell}(x)$. In this subsection we show how to approximate $P_{\alpha,\ell}(x)$. The main tool in the approximation is the following lemma.

LEMMA 3.3. *For any polynomial $P(x)$, α and ℓ, such that $0 \le \alpha \le 2^\ell$, then*

$$P_{\alpha,\ell}(\omega_d^k) = \frac{1}{2^\ell} \sum_{m=0}^{2^\ell-1} P\left(\omega_d^{k+m\frac{d}{2^\ell}}\right) \omega_d^{-\alpha m \frac{d}{2^\ell}}.$$

Proof. The proof is somewhat technical. We basically transform the right-hand side to the left-hand side. We start by substituting for $P(x)$ the expression $\sum_{j=0}^{d} a_j x^j$. This means that the right-hand side is

$$\frac{1}{2^\ell} \sum_{m=0}^{2^\ell-1} \sum_{j=0}^{d-1} a_j \omega_d^{j(k+m\frac{d}{2^\ell})} \omega_d^{-\alpha m \frac{d}{2^\ell}}.$$

We can change the order of summation and rewrite it as

$$\sum_{j=0}^{d-1} a_j \omega_d^{jk} \left[\frac{1}{2^\ell} \sum_{m=0}^{2^\ell-1} \left(\omega_d^{(j-\alpha)\frac{d}{2^\ell}}\right)^m\right].$$

Consider the expression between the brackets. If $j = \alpha \bmod 2^\ell$, then the expression is one, and otherwise it is equal to zero. Therefore we can rewrite it as

$$\sum_{j:\ j \bmod 2^\ell = \alpha} a_j \omega_d^{jk} = P_{\alpha,\ell}(\omega_d^k). \qquad \square$$

We can use Lemma 3.3 to approximate $P_{\alpha,\ell}$. Define the following random variable:

$$A_m(\alpha, \ell, k) = \frac{1}{m} \sum_{i=1}^{m} P\left(\omega_d^{k+\beta_i \frac{d}{2^\ell}}\right) \omega_d^{-\alpha\beta_i \frac{d}{2^\ell}},$$

where β_i is chosen uniformly in the set $\{0, \ldots, 2^\ell - 1\}$. The random variable $A_m(\alpha, \ell, k)$ is an approximation for $P_{\alpha,\ell}(\omega_d^k)$ that uses m sample points. The following lemma, which is an application of Chernoff bounds, states how "good" the approximation is.

LEMMA 3.4. *Let $P(x)$ be a polynomial such that $\max_j\{\|P(w_d^j)\|\} \le H$. There exists a constant c, such that for $m \ge cH^4 \log \frac{2}{\delta}$. Then, for any k,*

$$\text{Prob}\left[\|A_m(\alpha, \ell, k) - P_{\alpha,\ell}(\omega_d^k)\| \ge \frac{1}{16H}\right] \le \delta.$$

Proof. The proof is an application of the Chernoff bounds and the fact that $\|A_m(\alpha, \ell, k) - P_{\alpha,\ell}(\omega_d^k)\| \le 2H$. $\square$

The main use of Lemma 3.4 would be through the following claim.

CLAIM 3.5. *Let a and b be complex numbers such that $\|a\| \le H$. If $\|a - b\| \le \frac{1}{16H}$, then*

$$\left|\|a\|^2 - \|b\|^2\right| \le \frac{1}{4}.$$

3.3. Testing for zero polynomials. Our aim is to show that with high probability we can approximate each $ZERO(\alpha, \ell)$ correctly. This would imply that we can run the entire algorithm using the approximation, rather than testing $ZERO(\alpha, \ell)$. The approximating function is named $APPROX\_ZERO$. The main idea would be to approximate the power of the function. If the polynomial is the zero polynomial, then the power is zero, otherwise the power is at least one (since we assumed that the coefficients are integers).

For a polynomial $P(x) = \sum a_j x^j$, let $\text{power}(P) = \sum_j a_j^2$. In order to approximate the power of a polynomial, we define the following random variable:

$$B_{m_1,m_2}(\alpha, \ell) = \frac{1}{m_1} \sum_{i=1}^{m_1} \|A_{m_2}(\alpha, \ell, k_i)\|^2,$$

where k_i is chosen uniformly in the set $\{0, \ldots, d-1\}$. The random variable $B_{m_1,m_2}(\alpha, \ell)$ is an approximation of $\text{power}(P_{\alpha,\ell})$. The following lemma states how "good" the approximation is.

LEMMA 3.6. *Let $P(x)$ be a polynomial such that $H = \max_k\{\|P(w_d^k)\|\}$. There exists a constant c, such that for $m_1 \ge cH^4 \log \frac{1}{\delta}$ and $m_2 \ge cH^4 \log \frac{m_1}{\delta}$, then*

$$\text{Prob}\left[|B_{m_1,m_2}(\alpha, \ell) - \text{power}(P_{\alpha,\ell})| \ge \frac{1}{2}\right] \le \delta.$$

Proof. Lemma 3.3 shows how to compute $P_{\alpha,\ell}(\omega_d^k)$ from the values of $P(w_d^j)$. One can observe that the value of $P_{\alpha,\ell}(\omega_d^k)$ is a weighted average of the values of $P(w_d^j)$. From this we can deduce that $\max_k\{\|P_{\alpha,\ell}(\omega_d^k)\|\} \le H$.

The proof of the lemma uses Chernoff bounds. Since $\|P_{\alpha,\ell}(\omega_d^k)\|^2 \le H^2$, by choosing m_1 values of k_i's at random, such that $m_1 \ge cH^4 \log \frac{1}{\delta}$, Theorem 2.1 guarantees that with probability $1 - \delta/2$,

$$\left|\text{power}(P_{\alpha,\ell}) - \frac{1}{m_1} \sum_{i=1}^{m_1} \|P_{\alpha,\ell}(\omega_d^{k_i})\|^2\right| \le \frac{1}{4}.$$

Based on Lemma 3.4 for every k_i with probability $1 - \delta/(2m_1)$, the random variable $A_{m_2}(\alpha, \ell, k_i)$ approximates $P_{\alpha,\ell}(\omega_d^{k_i})$ with error less than $1/16H$. By Claim 3.5 this implies that the difference between $\|A_{m_2}(\alpha, \ell, k_i)\|^2$ and $\|P_{\alpha,\ell}(\omega_d^{k_i})\|^2$ is at most $1/4$, which completes the proof of the lemma. □

Now we are ready to define the function $APPROX\_ZERO(\alpha, \ell)$. In order to compute $APPROX\_ZERO(\alpha, \ell)$, we first compute $B_{m_1,m_2}(\alpha, \ell) = b$, where m_1 and m_2 are chosen according to Lemma 3.6, then if b is smaller than $1/2$ we return $TRUE$, otherwise we return $FALSE$. The following theorem states that with high probability, this is a good approximation.

THEOREM 3.7. *For any ℓ and α, $0 \leq \alpha \leq 2^{\ell}$,*

$$\text{Prob}[APPROX\_ZERO(\alpha, \ell) \neq ZERO(\alpha, \ell)] \leq \delta.$$

Proof. It is easy to show that if $APPROX\_ZERO(\alpha, \ell) \neq ZERO(\alpha, \ell)$, then

$$|B_{m_1,m_2}(\alpha, \ell) - \text{power}(P_{\alpha,\ell})| \geq \frac{1}{2};$$

the theorem follows from Lemma 3.6. □

3.4. Approximating a single coefficient. We need to approximate each coefficient that we found at the end of the algorithm with an error of at most ϵ. Given an exponent e, we show how to approximate its coefficients, a_e.

Define the random variable $COEF_m(e)$ to be

$$COEF_m(e) = \left\lfloor \frac{1}{2} + \frac{1}{m} \sum_{i=1}^{m} P(\omega_d^{\beta_i}) \omega_d^{-\beta_i e} \right\rfloor,$$

where β_i is chosen uniformly from $\{0, \ldots, d-1\}$. The random variable $COEF_m(e)$ is an approximation of the coefficient a_e using m sample points. The following lemma states how close the random variable $COEF_m(e)$ is to the coefficient a_e.

LEMMA 3.8. *Let P be a polynomial and $H = \max_j\{\|P(\omega_d^j)\|\}$, then for $m \geq 32H^2 \log \frac{2}{\delta}$*

$$\text{Prob}\,[COEF_m(e) \neq a_e] \leq \delta.$$

Proof. If $COEF_m(e) \neq a_e$ then $|COEF_m(e) - a_e| \geq 1$, since they are both integers. In such a case, $|\frac{1}{2} + \frac{1}{m}\sum_{i=1}^{m} P(\omega_d^{\beta_i})\omega_d^{-\beta_i e} - a_e| \geq 1/2$ The claim of the lemma follows from Chernoff bounds. □

3.5. Univariate interpolation algorithm: Small coefficients. We can now use the building blocks in this section to show the following result.

THEOREM 3.9. *Let P be a nonzero univariate polynomial with at most t integer coefficients, each of magnitude at most L, and $H = \max_k\{\|P(w_d^k)\|\}$. The running time of the algorithm is $O(H^8 t \log d \log^2 \frac{1}{\delta'} + tH^2 \log \frac{1}{\delta}) = \text{poly}(n, t, \log d, \log L)$, where $\delta' = O(\frac{1}{H^8 t \log d \log^2 \frac{1}{\delta}})$.*

Proof. There are $O(t \log d)$ recursive calls. Each time in $APPROX\_ZERO$ we test $O(H^4 \ln \frac{1}{\delta})$ different k_i's. For each k_i we use $O(H^4 \log \frac{1}{\delta})$ points to approximate it. At the end, for each of the t coefficients we call the procedure $COEF$ and use $O(H^2 \log \frac{1}{\delta})$ points. □

The main drawback of the above result is that the running time depends on H, which could be proportional to the largest coefficient, i.e., L, and not polynomial in the bit representation of the coefficients, i.e., $\log L$. In the next subsection we show how to overcome this deficiency.

4. Handling large coefficients. The aim of this section is to modify the algorithm of the previous section so that its running time would be polynomial in $\log L$. The technique

we use in handling large coefficients is similar in spirit to the scaling techniques in graph algorithms. We first find the exponents whose coefficients are "very" large. We approximate the coefficients of those exponents and subtract them from the polynomial. Since we only approximate the coefficients, the new polynomial may still have nonzero coefficients for those exponents. The main advantage is that the new polynomial has, in the worst case, the same number of nonzero coefficients, while the maximum value of a coefficient is smaller.

Consider a t-sparse polynomial $P(x)$, i.e.,

$$P(x) = \sum_{j=1}^{t} a_{e_j} x^{e_j}.$$

As before, let the parameter d be the smallest power of two larger than the degree, i.e., $d = 2^{\lceil \log(1+\max_j\{e_j\})\rceil}$. Similarly, let L be the smallest power of two that is larger than the maximum coefficient in absolute value, i.e., $L = 2^{\lceil \log(\max_j\{|a_{e_j}|\})\rceil}$.

Our aim is to perform the interpolation in time that is polynomial in t, $\log d$ and $\log L$. The following is the basic idea of the algorithm. Given L, the bound on the largest coefficient, we consider the sets $Z_1 = \{e_j \ : \ L/2 \leq |a_{e_j}| \leq L\}$ and $Z_2 = \{e_j \ : \ \frac{L}{8\sqrt{t}} \leq |a_{e_j}| \leq L\}$. We find a set Y such that $Z_1 \subset Y \subset Z_2$. Once we found the set Y (in time polynomial in t, $\log d$ and $\log L$), we approximate each of the coefficients of the exponents in Y with an absolute error of less than a $L/4$. This implies that, with high probability, the difference between the real coefficient and the approximated coefficient is less than $L/4$. For each $e_j \in Y$, denote by γ_{e_j} the approximated value of the coefficient a_{e_j} and in order to ensure that the new polynomial is integral, enforce γ_{e_j} to be integral. More formally, let Q be the polynomial we created,

$$Q(x) = \sum_{e_j \in Y} \gamma_{e_j} x^{e_j}.$$

Consider the polynomial $P'(x) = P(x) - Q(x)$. Assume that $Z_1 \subset Y \subset Z_2$ and $|\gamma_{e_j} - a_{e_j}| \leq L/4$, the polynomial P' is t-sparse, and has integer coefficients of size at most $L/2$. This implies that in polynomial time we reduced the problem of recovering the coefficients of a polynomial with maximum coefficient L (i.e., $P(x)$), to the problem of recovering the coefficients whose maximum coefficient is $L/2$ (i.e., $P'(x)$). Therefore after $O(\log L)$ such iterations the maximum coefficient is $O(\sqrt{t})$. Once the coefficients are of size $O(\sqrt{t})$ the algorithm of Theorem 3.9 would recover all the coefficients.

4.1. Approximating the power. The main idea is that with a sample size that depends on t and is independent of L, we can approximate power$(P_{\alpha,\ell})$. We start by modifying the approximation of $P_{\alpha,\ell}$.

LEMMA 4.1. *Let P be a t-sparse polynomial each of whose coefficient is bounded in magnitude by L. There exists a constant c, such that if $m \geq ct^4 \log \frac{2}{\delta}$, then for any k,*

$$\text{Prob}\left[\|A_m(\alpha, \ell, k) - P_{\alpha,\ell}(\omega_d^k)\| \geq \frac{L}{64t}\right] \leq \delta.$$

As before, this implies the following about the difference in the magnitude.

CLAIM 4.2. *Let a and b be complex numbers such that $\|a\| \leq tL$. If $\|a - b\| \leq \frac{L}{64t}$, then*

$$\left|\|a\|^2 - \|b\|^2\right| \leq \frac{L^2}{16}.$$

We can now claim the result about approximating the power.

LEMMA 4.3. *Let P be a t-sparse polynomial each of whose coefficient is bounded in magnitude by L. There exists a constant c such that if $m_1 \geq ct^4 \log \frac{1}{\delta}$ and $m_2 \geq ct^4 \log \frac{m_1}{\delta}$, then*

$$\text{Prob}\left[|B_{m_1,m_2}(\alpha, \ell) - \text{power}(P_{\alpha,\ell})| \leq \frac{3L^2}{32}\right] \leq \delta.$$

Proof. The polynomial P is t-sparse. Furthermore, each coefficient is bounded by L. Therefore, the magnitude of $P(w_d^k)$ is bounded by tL, and the magnitude of $P_{\alpha,\ell}(w_d^k)$ is bounded by tL. Once we bounded the magnitude of $P_{\alpha,\ell}$ we can use the Chernoff bounds to claim that with probability $1 - \delta/2$ the difference between $\text{power}(P_{\alpha,\ell})$ and $\frac{1}{m_1}\sum_{1\leq j\leq m_1} \|P_{\alpha,\ell}(\omega_d^{k_j})\|^2$ is less than $L^2/32$.

Based on Lemma 4.1, for every k_i with probability $1 - \delta/(2m_1)$ the random variable $A_{m_2}(\alpha, \ell, k_i)$ approximates $P_{\alpha,\ell}(\omega_d^{k_i})$ with error less than $L/(64t)$. By Claim 4.2 this implies that the difference between $\|A_{m_2}(\alpha, \ell, k_i)\|^2$ and $\|P_{\alpha,\ell}(\omega_d^{k_i})\|^2$ is at most $L^2/16$. □

Now we define the equivalent of the $ZERO$ and $APPROX\_ZERO$ procedures. Let $LARGE(P, L)$, where P is a polynomial and L an integer, be $TRUE$ if and only if P has some coefficient a_j such that $|a_j| \geq L/2$. As before, we cannot compute $LARGE(P, L)$ directly, but we can approximate it. We approximate the value of $LARGE(P_{\alpha,\ell}, L)$ by $APPROX\_LARGE(\alpha, \ell, L)$ which is defined as follows: Compute $B_{m_1,m_2}(\alpha, \ell) = b$ and if $b \geq \frac{5L^2}{32}$ return $TRUE$, otherwise return $FALSE$.

LEMMA 4.4. *Let P be a t-sparse polynomial, each of whose coefficient is bounded in magnitude by L. For any α and ℓ, $0 \leq \alpha \leq 2^\ell$, then*

$$\text{Prob}[APPROX\_LARGE(\alpha, \ell, L) = FALSE \text{ and } LARGE(P_{\alpha,\ell}, L) = TRUE] \leq \delta.$$

Proof. Assume that $LARGE(P_{\alpha,\ell}, L)$ is $TRUE$. This implies that $\text{power}(P_{\alpha,\ell}) \geq L^2/4$, since some coefficient is larger than $L/2$. With probability $1 - \delta$,

$$|B_{m_1,m_2}(\alpha, \ell) - \text{power}(P_{\alpha,\ell})| \geq \frac{3L^2}{32}.$$

This implies that with probability $1 - \delta$ we have $B_{m_1,m_2}(\alpha, \ell) \geq \frac{5L^2}{32}$, which is equivalent to $APPROX\_LARGE(\alpha, \ell, L)$ being $TRUE$. □

LEMMA 4.5. *Let P be a t-sparse polynomial each of whose coefficient is bounded in magnitude by L. For any α and ℓ, $0 \leq \alpha \leq 2^\ell$, then*

$$\text{Prob}\left[APPROX\_LARGE(\alpha, \ell, L) = TRUE \text{ and } LARGE\left(P_{\alpha,\ell}, \frac{L}{4\sqrt{t}}\right) = FALSE\right] \leq \delta.$$

Proof. Assume that $LARGE(P_{\alpha,\ell}, \frac{L}{4\sqrt{t}})$ is $FALSE$. This implies that $\text{power}(P_{\alpha,\ell}) \leq L^2/64$, since no coefficient is larger than $\frac{L}{8\sqrt{t}}$. With probability $1 - \delta$,

$$|B_{m_1,m_2}(\alpha, \ell) - \text{power}(P_{\alpha,\ell})| \geq \frac{3L^2}{32}.$$

This implies that with probability at most δ, we have $B_{m_1,m_2}(\alpha, \ell) \geq \frac{5L^2}{32}$, which is equivalent to $APPROX\_LARGE(\alpha, \ell, L) = TRUE$. □

The above two lemmas show that if $APPROX\_LARGE(\alpha, \ell, L)$ is $TRUE$, then with high probability $P_{\alpha,\ell}$ is a nonzero polynomial (and even has a coefficient larger than $\frac{L}{8\sqrt{t}}$), and if

$P_{\alpha,\ell}$ has a coefficient larger than $L/2$, then with high probability $APPROX\_LARGE(\alpha, \ell, L)$ is $TRUE$.

Note that the number of recursive calls when we use the procedure $LARGE$ is bounded, as before, by $t \log d$, since every time $LARGE$ is $TRUE$, we are focusing on at least one of the t coefficients.

4.2. Approximating a single coefficient. The approximation of a single coefficient is done as in the case of small coefficients, in §3.4. The only difference is that we need to approximate each coefficient that we found with absolute error less than $L/4$.

LEMMA 4.6. *There is a constant c such that for $m \geq ct^2 \log \frac{1}{\delta}$,*

$$\text{Prob}\,[|COEF_m(e) - a_e| \geq L/4] \leq \delta.$$

4.3. Putting it all together. The basic building block would be a procedure $FIND(P, L)$ that receives a polynomial P with coefficients of magnitude at most L and returns a list of exponents and their respective coefficients such that with high probability this list includes all the coefficients larger than $L/2$ and no coefficient smaller than $\frac{L}{8\sqrt{t}}$.

The procedure $FIND(P, L)$ works as the algorithm in §3, but uses the subroutine $APPROX\_LARGE$ instead of $APPROX\_ZERO$. At the beginning we start with a set Y_0 that has only 0. At phase ℓ, for each element $\alpha \in Y_\ell$, we create $\alpha_0 = \alpha$ and $\alpha_1 = \alpha + 2^\ell$. We add α_σ to $Y_{\ell+1}$ only if $APPROX\_LARGE(\alpha_\sigma, \ell + 1, L)$ is $TRUE$.

Since we are modifying the original polynomial, we describe how to compute the output that the black box returns to the algorithm. We have a polynomial $R(x)$, which is initially identically zero. (This polynomial would be used to "change" the original polynomial P that the algorithm is approximating.) Each time the algorithm queries the black box on a point ω_d^k, we query the original black box on input (d, k) and get a reply $\rho(d, k)$ (which is simply $P(\omega_d^k)$ with finite accuracy), we reply to the algorithm with $\rho(d, k) - R(\omega_d^k)$.

The main algorithm runs in phases. We maintain that after the ith phase, with high probability, $P(x) - R(x)$ is a polynomial with at most t nonzero coefficients and its maximum coefficient is at most $L/2^i$. The ith phase consists from the following steps: (1) We call $FIND(P - R, L/2^{i-1})$ that returns a set Y_i. With high probability the set Y_i includes all the coefficients in the range $L/2^i$ to $L/2^{i-1}$. (2) For each exponent in $e \in Y_i$ we approximate its coefficient a_e by γ_e, such that the difference is at most $L/2^{i+1}$. Let $Q_i(x) = \sum_{e \in Y_i} \gamma_e x^e$. (3) We add the polynomial $Q_i(x)$ to the polynomial $R(x)$.

Once the size of the coefficients is $O(\sqrt{t})$, the algorithm uses the algorithm for small coefficients (see Theorem 3.9) to completely recover the coefficients and adds this polynomial to R. At this point, with high probability, the polynomial $R(x)$ at is identical to the polynomial $P(x)$.

We have established the following theorem.

THEOREM 4.7. *There exists a randomized algorithm that interpolates a t-sparse polynomial, whose coefficients are integers less than L. The algorithm runs in time polynomial in t,* $\log d$, *and* $\log L$.

5. Precision accuracy. Our algorithms use complex numbers. When we measure the bit complexity, we need to specify the accuracy to which we require the complex numbers to be. This in turn also determines the bit complexity of the algorithm, since it determines the size of the numbers that the algorithm operates on. Since most of our operations are simply averaging over a set of random inputs, the algorithm requires only a low accuracy. The following claim shows how we can bound the precision accuracy.

CLAIM 5.1. *Adding m numbers with precision accuracy ϵ/m results in an output that has an absolute error of at most ϵ. Furthermore, if each of the m numbers is bounded in*

absolute value by H, then their description requires only $O(\log H + \log \frac{m}{\epsilon})$ bits to guarantee a precision accuracy of ϵ in the result.

The values used in $COEF_m(e)$ and $A_m(\alpha, \ell, k)$ are bounded by tL. This implies that by using $O(\log tL + \log \frac{m}{\epsilon})$ bits (and precision accuracy ϵ/m) in their computation, we introduce an additive error of at most $O(\epsilon)$. Since those random variables are used only as an approximation, the additional error could be absorbed in the error margins that we assumed before. (Recall that $B_{m_1,m_2}(\alpha, \ell)$ is define through A_m, and therefore the precision accuracy of B_{m_1,m_2} is determined by the precision accuracy of A_m.)

6. Approximation by polynomials. Assume that rather than recovering all the coefficients of the polynomial we are interested in recovering all the "large" coefficients of the polynomial. This is a very interesting case, since in many cases we may regard the other coefficients as "noise" that is not interesting.

The formal scenario is the following. As before, we are given a polynomial P as a black box, however, we are not guaranteed that it is either t-sparse or has integer coefficients. We assume that there is some polynomial Q, with t integer coefficients, such that the power of $P - Q$ is at most ν.

Our algorithm will recover a polynomial Q' such that the power of $P - Q'$ is bounded by $\nu + O(\epsilon)$. The number of recursive calls done by the algorithm is bounded by $O((t + \nu) \log d)$. The correctness is argued as before (each coefficient larger than one would be reached, with high probability). The running time can be bounded by observing that the difference between P and Q can cause the algorithm, for each specific value of ℓ, to search at most $O(\nu)$ values that would not lead to a coefficient of size at least one. The time to perform a single $APPROX\_ZERO$ or $APPROX\_LARGE$ remains the same.

We made a restriction that the approximating polynomial Q has integer coefficients. This restriction can be easily removed in the following way. Assume that there is a polynomial R that has real coefficients, and the power of $P - R$ is at most ν. Consider the polynomial R' that is achieved by multiplying the coefficients of R by t/ϵ, and rounding them to the nearest integer. The difference between $R(x)$ and $R'(x) * (\epsilon/t)$ is bounded by ϵ. This implies that the polynomial $P(x) * (t/\epsilon)$ has a good approximation by an integer polynomial R'.

7. Multivariate polynomials. The technique that we developed for univariate polynomials extends to multivariate polynomials and uses ideas similar to [GL89], [KM91], and [Zip79]. In this section we give a sketch of the ideas of the extension.

The idea for multivariate polynomials is the following. As before, we first recover all the "large coefficients." We proceed variable by variable, and find the exponents of the "large coefficients." Then we approximate the coefficients, and subtract them from the original polynomial. The main difference is that we recover the exponents of the interesting coefficients sequentially.

In the rest of this subsection we set the required definitions for the algorithm, and the test function that is used. Consider the polynomial

$$P(x_1, \ldots, x_n) = \sum_{j=1}^{t} a_{e_{1,j},\ldots,e_{n,j}} x_1^{e_{1,j}} x_2^{e_{2,j}} \cdots x_n^{e_{n,j}}.$$

As in the case of a single variable, we can compute a single coefficient as follows:

$$a_{e_1,\ldots,e_n} = \frac{1}{d^n} \sum_{k_1=0}^{d-1} \cdots \sum_{k_n=0}^{d-1} P(\omega_d^{k_1}, \omega_d^{k_2}, \ldots, \omega_d^{k_n}) \omega_d^{-k_1 e_1} \cdots \omega_d^{-k_n e_n}.$$

Also, the Parseval identity holds:

$$\sum_{j=1}^{t} a_{e_{1,j},\ldots,e_{n,j}}^2 = \frac{1}{d^n} \sum_{k_1=0}^{d-1} \cdots \sum_{k_n=0}^{d-1} \|P(\omega_d^{k_1}, \omega_d^{k_2}, \ldots, \omega_d^{k_n})\|^2.$$

The polynomial $P_{\alpha,\ell}$ is replaced by $P_{<e_1,\ldots,e_{j-1}>,\alpha,\ell}$, which is the polynomial that includes only the coefficients in which the exponents of $x_1, \ldots, x_{j-1}$ are $e_1, \ldots e_{j-1}$, respectively, the power of x_j equals to α modulus 2^ℓ, and the exponents of the variables $x_{j+1}, \ldots, x_n$ are unrestricted. The idea is that we are executing the univariate polynomial algorithm on the jth input. The input $x_1, \ldots x_{j-1}$ can appear with only a specific power, while there is no restriction on the exponents of $x_{j+1}, \ldots, x_n$. Formally,

$$P_{<e_1,\ldots,e_{j-1}>,\alpha,\ell}(x_1, \cdots, x_n) = x_1^{e_1} \cdots x_{j-1}^{e_{j-1}} \sum_{e_j \bmod 2^\ell = \alpha} x_j^{e_j} \sum_{e_{j+1}=0}^{d-1} \cdots \sum_{e_n=0}^{d-1} a_{e_1,\ldots,e_n} x_{j+1}^{e_{j+1}} \cdots x_n^{e_n}.$$

We define POWER as follows:

$$POWER[e_1, \ldots, e_{j-1}, \alpha, \ell] = E_{\vec{k}}[\| P_{<e_1,\ldots,e_{j-1}>,\alpha,\ell}(\vec{\omega_d^k})\|^2].$$

Where $\vec{k} = k_1, \ldots, k_n$ and $k_j \in \{0, \ldots, d-1\}$, and $\vec{\omega_d^k}$ is the input $x_1 = \omega_d^{k_1}, \ldots, x_n = \omega_d^{k_n}$. The expectation is uniform over all $\vec{k}$. As before, $POWER[e_1, \ldots, e_{j-1}, \alpha, \ell]$ equals the sum of the squares of coefficients in the polynomial $P_{<e_1,\ldots,e_{j-1}>,\alpha,\ell}(x_1, \ldots, x_n)$.

We search the jth variable before we continue to the $j+1$ variable. We find all the exponents to be prefixes of all the nonzero coefficients, up to the jth component (there are at most t). With each such prefix we continue to search for the $j+1$th component. The argument for correctness is similar to the argument for the univariate polynomials.

Consider the following example: $P(x_1, x_2) = q_1(x_1) + q_2(x_1)x_2$, where q_1 and q_2 are polynomials. Consider the expression

$$\begin{aligned} E_{k_2}\left[\left(q_1(\omega_d^k) + q_2(\omega_d^k)\omega_d^{k_2}\right)\left(q_1(\omega_d^{-k}) + q_2(\omega_d^{-k})\omega_d^{-k_2}\right)\right] &= E_{k_2}[q_1(\omega_d^k)q_1(\omega_d^{-k})] \\ &+ E_{k_2}[q_2(\omega_d^k)q_2(\omega_d^{-k})] \\ &+ E_{k_2}[q_1(\omega_d^k)q_2(\omega_d^{-k})\omega_d^{-k_2}] \\ &+ E_{k_2}[q_2(\omega_d^k)q_1(\omega_d^{-k})\omega_d^{k_2}]. \end{aligned}$$

Note that the first two terms do not depend on x_2 while the last two terms evaluate to zero, due to the averaging over x_2. In general, the averaging over all possible values of a certain variable causes the cross terms in the expression of the power to cancel out, therefore it has the effect of essentially "ignoring" this variable.

Acknowledgments. I would like to thank László Babai, Persi Diaconis, Marek Karpinski, Eyal Kushilevitz, Michael Rabin, and Prasoon Tiwari, for the discussions that I had with each of them.

REFERENCES

[BOT88] M. Ben-Or and P. Tiwari, *A deterministic algorithm for sparse multivariate polynomial interpolation*, in Proceedings of the 20th Annual ACM Symposium on Theory of Computing, pp. 301–309, ACM Press, New York, May 1988.

[GK87] D. Y. Grigoriev and M. Karpinski, *The matching problem for bipartite graphs with polynomially bounded permanents is in nc*, in 28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, pp. 166–172, October 1987, IEEE Press.

[GKS90a] D. Y. Grigoriev, M. Karpinski, and M. Singer, *Fast parallel algorithms for sparse multivariate polynomial interpolation over finite fields*, SIAM J. Comput., 19 (1990), pp. 1059–1063.

[GKS90b] ———, *Interpolation of sparse rational functions without knowing bounds on exponents*, in 31st Annual Symposium on Foundations of Computer Science, St. Louis, pp. 840–846, October 1990, IEEE Press.

[GL89] O. GOLDREICH AND L. LEVIN, *A hard-core predicate for all one-way functions*, in Proc. 21st ACM Symposium on Theory of Computing, pp. 25–32. ACM Press, New York, 1989.

[HR89] T. HAGERUP AND C. RUB, *A guided tour to chernoff bounds*, Inform. Process. Lett., 33 (1989), pp. 305–308.

[Kar89] M. KARPINSKI, *Boolean circuit complexity of algebraic interpolation problems*, Tech. Rep. TR-89-027, International Computer Science Institute, Berkeley, CA, 1989.

[KM91] E. KUSHILEVITZ AND Y. MANSOUR, *Learning decision trees using the fourier spectrum*, in Proceedings of STOC '91, pp. 455–464, ACM Press, New York, 1991.

[Zip79] R. E. ZIPPEL, *Probabilistic algorithms for sparse polynomials*, in EUROSAM, pp. 216–226. Lecture Notes Comput. Sci., 72, Springer-Verlag, New York, 1979, pp. 216–226.

SIAM J. COMPUT.
Vol. 24, No. 2, pp. 369–375, April 1995

012

AN $\mathcal{NC}$ ALGORITHM FOR EVALUATING MONOTONE PLANAR CIRCUITS*

A. L. DELCHER† AND S. RAO KOSARAJU‡

Abstract. Goldschlager first established that a special case of the monotone planar circuit problem can be solved by a Turing machine in $O(\log^2 n)$ space. Subsequently, Dymond and Cook refined the argument and proved that the same class can be evaluated in $O(\log^2 n)$ time with a polynomial number of processors. In this paper, we prove that the general monotone planar circuit value problem can be evaluated in $O(\log^4 n)$ time with a polynomial number of processors, settling an open problem posed by Goldschlager and Parberry.

Key words. parallel algorithms, planar circuits, monotone circuits

AMS subject classification. 68Q22

1. Introduction. The complexity of computing the output values of a boolean combinational circuit has been a much-studied problem in theoretical computer science. Ladner [12] showed that the general circuit value problem is $\mathcal{P}$-complete, and Goldschlager [4] showed that even if the circuit is restricted to be planar, or if it is restricted to be monotone, the problem remains $\mathcal{P}$-complete.

Goldschlager also established that a special case of the monotone planar circuit value problem can be solved by a Turing machine in $O(\log^2 n)$ space [5]. Subsequently, Dymond and Cook refined the argument and proved that the same class is in $\mathcal{NC}$ by showing that it can be evaluated in $O(\log^2 n)$ time with a polynomial number of processors [2]. In this paper, we prove that on a CREW PRAM the general monotone planar circuit value problem can be evaluated in $O(\log^4 n)$ time with a polynomial number of processors, which settles an open problem posed in [6].

In the next section we define the problem precisely. In §§3 and 4 we describe solutions to restricted versions of the problem. Finally, in §5 we show how the general problem can be reduced to these restricted versions.

2. Preliminaries. The input to our problem is a monotone combinational circuit, with a given embedding in the two-dimensional plane. Each node in the circuit is one of the following:

1. A boolean AND gate.
2. A boolean OR gate.
3. A NO-OP gate which has fan-in one and computes the identity function.
4. A constant gate with value 0 or 1. Such a node may have nonzero fan-in; however, the output is the specified constant, independent of the inputs.

Our problem will be to compute the truth value of every node in the circuit.

In a natural way, we view the circuit as a directed acyclic graph. We shall refer to nodes with zero fan-in (which must be constant nodes) as *source nodes*. Similarly we shall refer to nodes in the circuit with zero fan-out as *sink nodes*.

Without loss of generality, we can assume the following.

1. No gate has fan-in greater than two. Otherwise, we can easily replace higher fan-in gates with a planar arrangement of fan-in-2 gates.

*Received by the editors January 28, 1992; accepted for publication (in revised form) October 28, 1993.

†Department of Computer Science, Loyola College, Baltimore, Maryland, 21210-2699. Supported by National Science Foundation grants #IRI-8809324 and #CCR-8804284 and National Science Foundation DARPA grant #CCR-8908092.

‡Department of Computer Science, Johns Hopkins University, Baltimore, Maryland 21218-2686. Supported by National Science Foundation grant CCR-9107293 and NSF DARPA grant CCR-8908092.

2. The sink nodes of the circuit all lie on a single face of the planar embedding. Otherwise, in parallel for each separate sink independently, we can solve the circuit value problem consisting of exactly those nodes from each of which there is a path to the given sink. This results in a single sink, and the value of each node is the same as its value in the original circuit. In what follows we consistently arrange the circuits so that the sinks lie on the outermost face of the embedding. Our algorithms, however, do not depend on this fact.

Note that all these assumptions can be realized from an arbitrary instance of the problem by an $\mathcal{NC}$ computation. For example, testing a graph for planarity and constructing its embedding can be done in $O(\log^2 n)$ time with $O(n)$ processors [10]. Reachability for general graphs can be computed by simple transitive closure in $O(\log^2 n)$ time with $O(n^3)$ processors, and for planar directed graphs in $O(\log^4 n)$ time with only $O(n)$ processors [8]. Further references may be found in [1], [3], [7], [9], and [14].

We shall describe the algorithm for the general problem by presenting algorithms for a sequence of special cases to which the general problem can be reduced. Throughout this paper, unless otherwise specified, the term "circuit" should be interpreted to mean a monotone circuit with a given planar embedding. This embedding will remain fixed throughout all the computations.

3. Stratified circuits. We begin with a special class of circuits in which all nodes are arranged into levels such that all interconnections occur in a regular fashion between adjacent levels.

Let $C = (V, E)$ be a circuit. We use $|C|$ to denote the number of nodes in C. By the *level* of any node v in C we mean the length of its longest path to a sink node. We assume that level numbers are calculated for each node based on the initial circuit C, and that these same level numbers are used in all recursive procedures on subcircuits of C. Thus, the level of each sink node in C is 0, but in subcircuits of C, sink nodes may have other level numbers. We use L_C to denote the highest-numbered level in C. We shall refer to nodes at level L_C of the circuit as *primary source nodes*. (Note that all nodes at level L_C must be source nodes, since otherwise there would be a level number higher than L_C.) Finally, let $C_{i..j}$ denote the subcircuit of C that comprises all nodes at levels i through j.

DEFINITION 3.1. *A* stratified circuit *is a circuit $C = (V, E)$ in which we have the following items.*

1. *Every arc connects nodes at consecutive level numbers, i.e., for each arc $u \to v$ in E, $\mathit{level}(u) = 1 + \mathit{level}(v)$.*

2 (*Nested level property*). *For $0 \le i \le L_C$, all level-i nodes in $C_{0..i}$ lie on a single face. (Recall that the planar embedding for $C_{0..i}$ is fixed by the given embedding of C.)*

3. *Every source node is a primary source node.*

A restricted stratified circuit *is a stratified circuit in which all constant gates have fan-in zero.*

Dymond and Cook showed that a restricted stratified circuit with left and right boundaries for each level could be solved in $O(\log^2 n)$ time with a polynomial number of processors [2]. Their algorithm computes truth values at each level of the circuit as alternating intervals of 0/1 values. By determining where the alternations occur, the truth values of all remaining nodes are implicitly computed. The algorithms in [5] and [2] rely crucially on the fact that in a monotone planar circuit, no alternation at level i of the circuit can result in more than one alternation at level $i - 1$.

We note that the Dymond–Cook algorithm does not require that all sinks of the circuit be on the same face, or that level numbers be defined in terms of distance to sink nodes. In particular, if u is an AND- or OR-node in a restricted stratified circuit, the result of converting u to a NO-OP-node and deleting one of its input connections remains a restricted stratified circuit, although a new sink node may have been created.

The algorithm of Dymond and Cook [2] can be modified in a straightforward fashion to handle any restricted stratified circuit. The only difference is that whereas each level of the circuit in the Dymond–Cook case has distinct left and right boundaries, in the general restricted stratified circuit each level is a "ring." These rings have the same properties with regard to alternations, however, and the alternations can be computed in the same fashion, simply by allowing for the wraparound. The modifications needed to the algorithm of Dymond and Cook are minor and obvious.

Thus, we have the next proposition.

PROPOSITION 3.2. *The value at each node in a restricted stratified circuit can be computed in* $O(\log^2 n)$ *time with a polynomial number of processors.*

Following the approach in [11], we generalize slightly our notion of stratified circuits as follows in Definition 3.3.

DEFINITION 3.3. *A* stratified circuit with variables *is a stratified circuit in which primary source nodes, in addition to being labelled* 0 *or* 1, *may be labelled by a third possibility,* $\boxed{?}$, *indicating that the value of the node is not known.*

A restricted stratified circuit with variables *is a stratified circuit with variables in which all constant gates have fan-in zero.*

We shall refer to source nodes labelled with $\boxed{?}$ as *variables*. The value to be computed at each node v in circuit with variables now also has three possibilities: 0 or 1 if the value of v is independent of any variable inputs, or $\boxed{?}$ if v does depend on variable nodes, i.e., if there is at least one assignment of 0/1 values to variables that makes v be 0, and at least one assignment that makes v be 1. Note that our initial problem is the special case of this generalized problem in which there are no input variables, so that the value of no node in the circuit is $\boxed{?}$.

PROPOSITION 3.4. *The value at each node in a restricted stratified circuit with variables can be computed in* $O(\log^2 n)$ *time with a polynomial number of processors.*

Proof. The version with variables can be reduced to the version without variables as follows.

Step 1. Set the value of all source variables to 0 and compute the value of each node in the resulting restricted stratified circuit. Any node whose value is 1 will be 1 for any assignment of values to the input variables.

Step 2. Set the value of all source variables to 1 and compute the value of each node in the resulting restricted stratified circuit. Any node whose value is 0 will be 0 for any assignment of values to the input variables.

Step 3. All other nodes depend on the input variables and should be labelled $\boxed{?}$.

The correctness of this procedure follows from the fact that the circuit is monotone, containing only AND and OR operations. By Proposition 3.2, each step can be performed in $O(\log^2 n)$ time with a polynomial number of processors. □

We now consider the entire class of stratified circuits with variables. Our strategy in evaluating such a circuit is to split the circuit into two subcircuits, C_1 and C_2, where the outputs of C_2 are the inputs to C_1. We change the inputs of C_1 to be variables and then recursively solve the two subcircuits. Finally, we convert C_1 into a restricted stratified circuit and resolve it by using the correct outputs of C_2.

More specifically, let $C = (V, E)$ be a stratified circuit with $|C| = n$. We determine the value at each node in C as follows.

Step 1. Find a level i such that $n/4 \leq |C_{0..i}| < 3n/4$ and $n/4 \leq |C_{i..L_C}| < 3n/4$. Such an i exists since the fan-in of each node in C is at most 2, if we assume that C is connected.[1] If C is not connected, we can compute its connected components and process each separately.

[1] When we say C is connected we mean that the graph obtained by making all arcs in C undirected is connected.

Step 2. Relabel all level-i AND, OR, and NO-OP nodes in $C_{0..i}$ to $\boxed{?}$, and denote the resulting circuit C_1. Let C_2 denote $C_{i..L_C}$.

Step 3. Recursively solve C_1 and C_2 in parallel. The values of the C_2 nodes are the correct values for that part of the original circuit C.

Step 4. Remove from C_1 all nodes whose labels have been determined to be 0 or 1, together with all out edges from these nodes. Note that the other end of such an edge is now either labelled 0 or 1 (and hence is removed), or else is labelled $\boxed{?}$ and has exactly one input also labelled $\boxed{?}$. Also convert AND and OR nodes labelled $\boxed{?}$ and having exactly one input labelled $\boxed{?}$ to a NO-OP node. Call the resulting circuit C_1'. Note that all in-degree-0 nodes of C_1' are at level i, i.e., C_1' is a restricted stratified circuit with variables.

Step 5. Assign the labels computed for the sink nodes of C_2 to the corresponding C_1' sources, and call the resulting circuit C_1''. Since C_1'' is a restricted, stratified circuit with variables, its values can be computed as described in Proposition 3.4.

Steps 1, 2, and 4 are easily accomplished in $O(\log n)$ time with a polynomial number of processors, and by Proposition 3.4, Step 5 can be computed in $O(\log^2 n)$ time with a polynomial number of processors. In Step 3, since there are two parallel recursive calls on circuits of size at most $3n/4$, the entire algorithm runs in $O(\log^3 n)$ time with a polynomial number of processors.

PROPOSITION 3.5. *The value at each node of a stratified circuit C (with or without variables) can be computed in $O(\log^3 n)$ time with a polynomial number of processors.*

4. Focused circuits. In this section we show how another class of circuits can be converted to stratified circuits, and therefore can be solved in $\mathcal{NC}$.

DEFINITION 4.1. *A* focused circuit *is a circuit $C = (V, E)$ in which there is a subset of the sources $\{s_1, \ldots, s_k\}$ lying on a single face of C such that every non-source node in C can be reached by a (directed) path from some s_i.*

Note that in a focused circuit, any node that cannot be reached from $\{s_1, \ldots, s_k\}$ must be a source node. Recall that we are assuming all circuits are given with a planar embedding that has all sinks on a single face.

We now describe a procedure that converts any given focused circuit C into an equivalent stratified circuit C'. By equivalent we mean that every node of C has a corresponding node in C' that computes the same value.

Procedure Convert

Step 1. For each node v, determine the length ℓ_v of the longest path from any s_i to v. Let $\ell_{\max}$ be the maximum value of ℓ_v in C.

Step 2. Set the level number of all sink nodes to 0, and set the level number of every other node v to $\ell_{\max} - \ell_v$.

Step 3. For each arc $u \to v$ of the circuit with $\mathit{level}(u) > \mathit{level}(v) + 1$, replace $u \to v$ with a path of length $\mathit{level}(u) - \mathit{level}(v)$ composed entirely of NO-OP nodes. Assign appropriate level numbers for these new nodes. (Note that $\mathit{level}(u)$ cannot be less than $\mathit{level}(v)$.)

Step 4. Remove every source node v that is not reachable from any s_i by making an appropriate change to the labels of nodes adjacent to v. For example, if the value of v is 1 and there is an arc $v \to u$, then make u a NO-OP node if it were an AND gate, make u a constant-1 node if it were an OR gate, and leave u unchanged if it is a constant gate. Note that u cannot be a NO-OP gate since such gates have fan-in of one and u is reachable from some s_i and is adjacent to source node v. A similar set of changes applies if the value of v is 0.

PROPOSITION 4.2. *The above procedure converts a focused circuit C into an equivalent stratified circuit C'.*

Proof. Clearly, C' is a planar circuit equivalent to C since the only changes made are the replacement of arcs by paths of NO-OP nodes and the partial evaluation of nodes with constant inputs. To show that C' is a stratified circuit, it is easy to verify that the procedure correctly determines level numbers, that every arc in C' connects nodes at successive levels, and that the only source nodes are $\{s_1, \ldots, s_k\}$, which are primary source nodes. It only remains to verify that C' satisfies the nested level property.

Suppose C' did not satisfy the nested level property. Then let i be a level number such that in $C'_{0..i}$ there are two level-i nodes, a and b, that do not lie on the same face. But a and b are both reachable from the face containing $s_1, \ldots, s_k$ by edges not contained in $C'_{0..i}$, which contradicts the planarity of C'. Thus, C' must satisfy the nested level property. □

Steps 2–4 in Procedure Convert are accomplished easily in $O(\log n)$ time with a polynomial number of processors. Step 1, finding the longest paths in a directed acyclic graph, can be accomplished by matrix multiplications by using addition and MAX operations in $O(\log^2 n)$ time with a polynomial number of processors [1], [3]. Thus, from Propositions 3.5 and 4.2 we have the next proposition.

PROPOSITION 4.3. *The value at each node of a focused monotone planar circuit C can be computed in $O(\log^3 n)$ time with a polynomial number of processors.*

5. General circuits. In this section we show how the procedures from the previous sections can be used to construct an $\mathcal{NC}$ algorithm to compute the value at each node in any monotone planar circuit C.

In $C = (V, E)$, with $|V| = n$, and for any set of nodes S, let ***reach***(S) denote the subgraph of C by comprising the union of all (directed) paths that begin at a node in S. By an abuse of notation we denote ***reach***$(\{s_i, \ldots, s_j\})$ as simply ***reach***$(s_i, \ldots, s_j)$. By a *connected component* of $C \setminus$ ***reach***(S), we mean a subcircuit of C, which is a connected component in the underlying undirected graph of C with ***reach***(S) having been removed. Let $s_1, s_2, \ldots, s_k$ be the source nodes of C.

We consider two cases.

Case 1. Some s_i, $1 \leq i \leq k$, has the property that no connected component of $C \setminus$ ***reach***(s_i) contains more than $n/2$ nodes.

In this case, we can first, in parallel, recursively solve the subcircuits corresponding to each connected component in $C \setminus$ ***reach***(s_i), and then use the results to solve the subcircuit corresponding to ***reach***(s_i), which is now a focused circuit.

Case 2. For each s_i, $1 \leq i \leq k$, some connected component of $C \setminus$ ***reach***(s_i) contains more than $n/2$ nodes.

Fix an arbitrary order for $s_1, s_2, \ldots, s_k$. We first find the smallest index i_0 such that $C \setminus$ ***reach***$(s_1, \ldots, s_{i_0})$ has a connected component with more than $n/2$ nodes, but that no connected component of $C \setminus$ ***reach***$(s_1, \ldots, s_{i_0}, s_{i_0+1})$ has more than $n/2$ nodes. Clearly such an i_0 exists, since every node in C is contained in ***reach***$(s_1, \ldots, s_k)$, and can be found within $O(\log^3 n)$ time by an $\mathcal{NC}$ algorithm.

Let P be the connected component of $C \setminus$ ***reach***$(s_1, \ldots, s_{i_0})$ that contains more than $n/2$ nodes. (See Fig. 1.) By our choice of i_0, s_{i_0+1} must lie within P. In addition, observe that there is no arc in C from a node u outside P to a node v in P, since otherwise ***reach***$(s_1, \ldots, s_{i_0})$ would include v. Note also that no connected component of $P \setminus$ ***reach***(s_{i_0+1}) contains more than $n/2$ nodes, since $C \setminus$ ***reach***$(s_1, \ldots, s_{i_0}, s_{i_0+1})$ has no connected component with more than $n/2$ nodes.

To solve the original circuit C, we first find index i_0 and connected component P as above. We then compute $C_1 =$ ***reach***(P) and $C_2 =$ ***reach***(s_{i_0+1}). Now recursively, in parallel, we

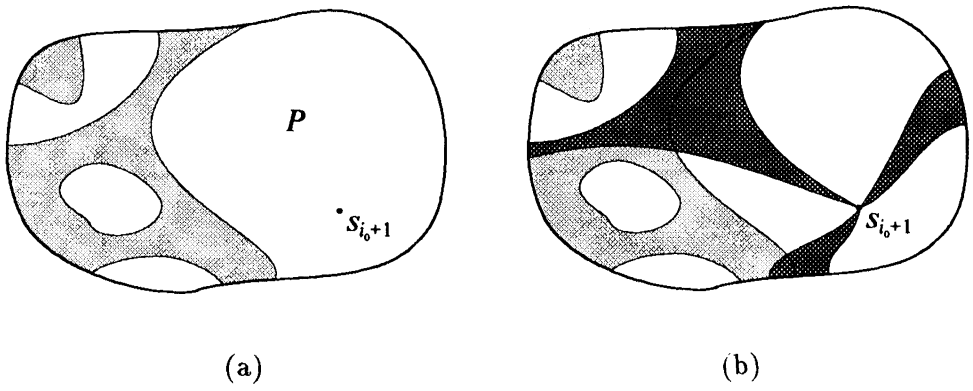


FIG. 1. *Abstract Representation of a Case* 2 *Circuit. Recall that all sinks lie on the outer face.* (a) *The shaded region represents* reach$(s_1, \ldots, s_{i_0})$. (b) *Nodes in* $C_2 =$ reach(s_{i_0+1}) *are now also shaded.*

solve all the subcircuits corresponding to connected components of both $C \setminus C_1$ and $P \setminus C_2$. Note that each of these subcircuits contains fewer than $n/2$ nodes. After all these subcircuits have been solved, what remains of the original circuit is $C_2 \cap P$ within P and $C_1 \setminus P$ outside of P. Now $C_2 \cap P$ can be solved as a focused circuit, since by definition every node in C_2 is reachable from s_{i_0+1}. Note that all inputs needed to solve $C_2 \cap P$ are now available as a result of solving $P \setminus C_2$. Finally, $C_1 \setminus P$ can be solved as a focused circuit, since all its nodes can be reached from source nodes on the face representing P, and the values of these source nodes have been computed. In addition, the previous solution of the connected components of $C \setminus C_1$ provides the other necessary inputs to $C_1 \setminus P$.

In both cases, by performing a series of $\mathcal{NC}$ computations in $O(\log^3 n)$ time, we reduce the problem to parallel recursive calls on subcircuits of size at most $n/2$, where the combined size of all these subcircuits is less than n.

Thus we show our main result in the following proposition.

PROPOSITION 5.1. *The value at each node of a monotone, planar circuit C can be computed in* $O(\log^4 n)$ *time with a polynomial number of processors.*

Recently, Yang [15] has developed an $O(\log^3 n)$-time $\mathcal{NC}$ algorithm for solving monotone planar circuits using the straight-line-code parallel evaluation technique of Miller, Ramachandran, and Kaltofen [13].

REFERENCES

[1] S. G. AKL, *The Design and Analysis of Parallel Algorithms*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[2] P. W. DYMOND AND S. A. COOK, *Hardware complexity and parallel computation*, in Proc. 21st IEEE Symposium on Foundations of Computer Science, 1980, pp. 360–372.

[3] A. GIBBONS AND W. RYTTER, *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge, UK, 1988.

[4] L. M. GOLDSCHLAGER, *The monotone and planar circuit value problems are log space complete for P*, SIGACT News, 9 (1977), pp. 25–29.

[5] ———, *A space efficient algorithm for the monotone planar circuit value problem*, Inform. Process. Lett., 10 (1980), pp. 25–27.

[6] L. M. GOLDSCHLAGER AND I. PARBERRY, *On the construction of parallel computers from various bases of boolean functions*, Theoret. Comput. Sci., (1986), pp. 43–58.

[7] J. JÁJÁ, *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA, 1992.

[8] M.-Y. KAO AND P. N. KLEIN, *Towards overcoming the transitive-closure bottleneck: Efficient parallel algorithms for planar digraphs*, in Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, 1990, pp. 181–192.

[9] R. M. KARP AND V. RAMACHANDRAN, *Parallel algorithms for shared-memory machines*, in Handbook of Theoretical Computer Science, J. Van Leeuwen, ed., North-Holland, Amsterdam, 1990, pp. 869–941.

[10] P. N. KLEIN AND J. H. REIF, *An efficient parallel algorithm for planarity*, J. Comput. System Sci., 37 (1988), pp. 190–246.

[11] S. R. KOSARAJU, *On the parallel evaluation of classes of circuits*, in Foundations of Software Technology and Theoretical Computer Science, K. V. Nori and C. E. Veni Madhavan, eds., Springer-Verlag, New York, 1990, pp. 232–237. LNCS No. 472.

[12] R. E. LADNER, *The circuit value problem is log-space complete for P*, SIGACT News, 7 (1975), pp. 18–20.

[13] G. L. MILLER, V. RAMACHANDRAN, AND E. KALTOFEN, *Efficient parallel evaluation of straight-line code and arithmetic circuits*, SIAM J. Comput., 17 (1988), pp. 687–695.

[14] J. H. REIF, ED., *Synthesis of Parallel Algorithms*, Morgan Kaufmann, San Mateo, CA, 1991.

[15] H. YANG, *An $\mathcal{NC}$ algorithm for the general planar monotone circuit value problem*, in Proc. 3rd IEEE Symposium on Parallel and Distributed Processing, 1991, pp. 196–203.

SIAM J. COMPUT.
Vol. 24, No. 2, pp. 376–386, April 1995

013

PRIVATE COMPUTATIONS OVER THE INTEGERS*

BENNY CHOR†, MIHÁLY GERÉB-GRAUS‡, AND EYAL KUSHILEVITZ§

Abstract. The subject of this work is the possibility of private distributed computations of n-argument functions defined over the integers. A function f is t-private if there exists a protocol for computing f, so that no coalition of at most t participants can infer any additional information from the execution of the protocol. It is known that over *finite* domains every function can be computed $\lfloor (n-1)/2 \rfloor$-privately. Some functions, like addition, are even n-private.

We prove that this result cannot be extended to infinite domains. The possibility of privately computing f is shown to be closely related to the *communication complexity* of f. By using this relation, we show, for example, that n-argument addition is $\lfloor (n-1)/2 \rfloor$-private over the nonnegative integers, but not even 1-private over all the integers.

Finally, a complete characterization of t-private *Boolean* functions over countable domains is given. A Boolean function is 1-private if and only if its communication complexity is *bounded*. This characterization enables us to prove that every Boolean function falls into one of the following three categories: It is either n-private, $\lfloor (n-1)/2 \rfloor$-private but not $\lceil n/2 \rceil$-private, or not 1-private.

Key words. private distributed computations, communication complexity

AMS subject classifications. 94A15, 94A60, 68R05

1. Introduction.

A set of $n \geq 3$ computationally unbounded parties, each holding an input x_i taken from a domain D, wishes to cooperate in distributively computing the value $f(x_1, x_2, \ldots, x_n)$ of a predetermined function f. These parties are honest, namely, they all follow their prescribed protocol. They communicate over a complete point-to-point communication network, where eavesdropping is not possible. A function f is called *t-private* if there is a communication protocol for computing f so that no coalition of at most t participants gets any additional information from the execution of the protocol. Ben-Or, Goldwasser, and Wigderson [5] and Chaum, Crépeau, and Dåmgard [8] have shown that if the domain D of f is *finite*, then f is $\lfloor (n-1)/2 \rfloor$-private. Specific functions, like addition, are even n-private over finite domains [6], while certain functions, like Boolean OR, are not $\lceil n/2 \rceil$-private [5].[1] However, functions of interest are typically defined not over finite domains, but over all strings, over the integers, or more generally over some countable domain. To apply the protocol of [5] and [8], one has to (implicitly or explicitly) assume an upper bound on the input size. If the bound does not hold, then the protocol, which depends not only on f but also on the size of the domain, has to be adjusted. This adjustment amounts to revealing additional information on the magnitude of the inputs.

The question we deal with in this work is private computations of functions defined over countable domains. In other words, is there a private protocol for computing f which can be applied to all inputs, regardless of their sizes? Before going any further, we remark that the $\lfloor (n-1)/2 \rfloor$-private protocol of [5] and [8] cannot be used in the infinite domain. Its first step is to apply a secret sharing scheme to every input. Secret sharing schemes strongly rely on the finiteness of the domain, and indeed do not exist over countable domains [7], [11].

*Received by the editors February 1, 1991; accepted for publication (in revised form) November 2, 1993. An early version of this paper appeared in *Proc. of 31st IEEE Symposium on Foundations of Computer Science*, 1990, pp. 335–344. This research was supported by US-Israel Binational Science Foundation grant 88-00282.

†Department of Computer Science, Technion, Haifa 32000, Israel (benny@cs.technion.ac.il). Part of this research was done while the author was visiting the Computer Science Department in the University of Toronto.

‡Department of Computer Science, Tufts University, Medford, Massachusetts 02155 (gereb@cs.tufts.edu).

§Department of Computer Science, Technion, Haifa 32000, Israel (eyalk@cs.technion.ac.il).

[1] The study of private distributed computing was initiated in [18] and [13]. The model used there is different – the communication lines are not secure against eavesdropping, and on the other hand the participants are computationally bounded, so cryptographic techniques can be employed.

We show that the privacy of f is closely related to the *communication complexity* of f as defined by Yao [17]; namely, to the number of bits that need to be communicated among n parties in order to compute f (using any protocol, not necessarily a private one). The connection between these two notions enables us to show that a wide class of functions are $\lfloor (n-1)/2 \rfloor$-private. This class includes, among others, maximum and minimum, addition over the nonnegative integers, and multiplication over all integers. The relations to communication complexity are further used in our impossibility results. For example, we show that over the (negative and nonnegative) integers, addition is not even 1-private. That is, there is no protocol which computes the sum of n integers and preserves privacy even with respect to single participants. This contrasts with the $\lfloor (n-1)/2 \rfloor$-privacy of addition over the nonnegative integers, and the n-privacy of addition over any finite domain.

We give a complete characterization of private Boolean functions: A Boolean function is 1-private if and only if its communication complexity is *bounded*. Since many Boolean functions have unbounded communication complexity, this proves that there are Boolean functions which are not even 1-private. Furthermore, we show that if a Boolean f is 1-private, then it is also $\lfloor (n-1)/2 \rfloor$-private. In [10] it was shown that if a Boolean f is $\lceil n/2 \rceil$-private, then it is also n-private. Hence there is a three-level privacy hierarchy for Boolean functions: Every Boolean function (defined over a countable domain) is either n-private, $\lfloor (n-1)/2 \rfloor$-private but not $\lceil n/2 \rceil$-private, or not 1-private.

The remainder of this paper is organized as follows: In §2 we describe the model and give the needed definitions. In §3 we apply communication complexity arguments to produce private protocols for various functions. Section 4 contains the impossibility results for the functions addition and integer gcd. In §5 we prove our characterization for the Boolean case, and its implications. Section 6 introduces a model of nonterminating protocols which compute f "in the limit," and demonstrate some intriguing private protocols in this model. Finally, §7 contains concluding remarks and open problems.

2. Model and definitions. The system consists of a synchronous network of n computationally unbounded parties, $P_1, P_2, \ldots, P_n$. Each pair of parties is connected by a secure (no eavesdropping) and reliable communication channel. At the beginning of an execution, each party P_i has an input $x_i \in \{0,1\}^*$. (No probability space is associated with the inputs.) In addition, each party has a random input r_i taken from a source of randomness R_i (the random inputs are independent). The parties wish to compute the value of a function $f(x_1, x_2, \ldots, x_n)$. To this end, they exchange messages as prescribed by a protocol $\mathcal{F}$. The parties are honest, that is, they follow the protocol $\mathcal{F}$, and there are no failures of any kind. Messages are sent in rounds, where in each round every processor sends a (nonempty) message to every other processor. The protocol specifies n functions, one per party, which determine the messages sent by each party. The arguments to the messages producing function of each processor are its input, its random input, the round number, the messages it received so far, and the identity of the receiver.

The communication S sent in an execution of the protocol is the concatenation of all messages sent in the execution, parsed according to sender, receiver, and round number. For a subset $T \subseteq \{1, 2, \ldots, n\}$, we denote by S_T the communication S with the exception of messages sent between parties in $\bar{T}$. The communication length of the protocol $\mathcal{F}$ on inputs $x_1, \ldots, x_n$ is the maximum length of all communications S, sent in $\mathcal{F}$ on these inputs, over all random inputs. We say that a communication string S, parsed as above, is *consistent* with the protocol $\mathcal{F}$, party P_i, and input x_i if the following holds: There is a positive probability that if P_i has input x_i and receives the messages in the string S that are destined to P_i, then P_i will send messages identical to those messages in S that emanate from P_i. (Notice that to be consistent with just one party, the communication string need not be an actual string sent in some execution.)

We say that a protocol $\mathcal{F}$ computes the function f with *ε-advantage* ($0 < \varepsilon \leq \frac{1}{2}$) if for every input $\vec{x} = (x_1, \ldots, x_n)$ the last message in the protocol, is an identical message sent by party P_1 to all parties, indicating that this is the last message in the execution, and containing a value $\mathcal{F}(\vec{x}, \vec{r})$ which satisfies

$$Pr_{\vec{r}}(\mathcal{F}(\vec{x}, \vec{r}) = f(\vec{x})) \geq \frac{1}{2} + \varepsilon .$$

Denote by $p_k(\vec{x})$ the probability that more than k bits are exchanged between the parties during the execution of the protocol $\mathcal{F}$ with input $\vec{x}$, (the probability space is over the random inputs of all parties). The protocol is *terminating* if for every input $\vec{x}$,

$$\lim_{k \to \infty} p_k(\vec{x}) = 0 .$$

We say that a coalition (i.e., a set of parties) T *does not learn any additional information* (other than what follows from its input and the function value) from the execution of a randomized protocol $\mathcal{F}$, which computes f, if the following holds: For every two inputs $\vec{x}, \vec{y} \in (\{0,1\}^*)^n$ that agree in their T entries (i.e., $\forall i \in T : x_i = y_i$) and satisfy $f(\vec{x}) = f(\vec{y})$, the messages passed between T and $\bar{T}$ are identically distributed. That is, for every communication S,

$$Pr_{\vec{r}}(S_T | \vec{x}) = Pr_{\vec{r}}(S_T | \vec{y}) ,$$

where the probability space is over the random inputs of all parties.

We say that a protocol $\mathcal{F}$ for computing f is *t-private* if any coalition T of at most t parties does not learn any additional information from the execution of the protocol. We say that a function f is *t-private* if there exists a t-private terminating protocol that computes it with ε-advantage, for some $0 < \varepsilon \leq \frac{1}{2}$.

We end this section with some standard communication complexity definitions. Let $\{0,1\}^{\leq m}$ denote the collection of binary strings whose length is at most m. The ε-advantage *communication complexity* of f, when restricted to $\{0,1\}^{\leq m}$ denoted $C_\varepsilon(f_m)$, is the minimum over all n-party protocols which compute f with ε-advantage over $\{0,1\}^{\leq m}$, of the worst case communication length of the protocol, over all n-tuples of inputs that are all at most m-bit long. (Notice that party P_i has input x_i. This is the "regular" definition, as in [17] and [16], and should not be confused with the one of Chandra, Furst and Lipton [9].) The ε-advantage *communication complexity* of a function f, denoted $C_\varepsilon(f)(m)$, is defined as $C_\varepsilon(f_m)$. For *deterministic* protocols, $C_{\det}(f)(m)$ is defined similarly. We say that f has *bounded* communication complexity if there is some positive ε ($0 < \varepsilon \leq 1/2$) and an integer d such that for all m, $C_\varepsilon(f)(m) \leq d$.

3. Functions that can be privately computed. In this section we present a sufficient condition, based on $C_\varepsilon(f)$, for $\lfloor (n-1)/2 \rfloor$-privacy of f. This enables us to derive $\lfloor (n-1)/2 \rfloor$-private error-free protocols for a wide family of "natural" functions. We start with a lemma.

LEMMA 3.1. *Let $B \subseteq \{0,1\}^*$ and let $f : (\{0,1\}^*)^n \to B$. If for every $b \in B$ the function*

$$f_b(\vec{x}) = \begin{cases} 1 & \text{if } f(\vec{x}) = b, \\ 0 & \text{otherwise,} \end{cases}$$

can be computed t-privately, using an error-free protocol, then f is t-private.

Proof. The value of $f_b(\vec{x})$ for every $b \in B$ is determined by the value of $f(\vec{x})$. Thus, to compute $f(\vec{x})$, the parties can go over every $b \in B$ (say, in lexicographic order) and compute $f_b(\vec{x})$ by using the given t-private protocol for f_b without revealing any additional

information. The protocol terminates when for some $b \in B$ the value of $f_b(\vec{x})$ is 1. From the definition of f_b, this implies that $f(\vec{x}) = b$. Since every function f_b is computed by an error-free, t-private protocol, it is not hard to verify that the resulting protocol is error-free and is t-private. □

THEOREM 3.2. *Let* $f : (\{0,1\}^*)^n \to B$ *and* $0 < \varepsilon \leq \frac{1}{2}$. *If for every* $b \in B$ *the communication complexity* $C_\varepsilon(f_b)$ *is bounded, then* f *is* $\lfloor (n-1)/2 \rfloor$*-private.*

Proof. Using Lemma 3.1, it is enough to prove that for every $b \in B$, the function f_b is $\lfloor (n-1)/2 \rfloor$-private. Fix $b \in B$. By the assumption, $C_\varepsilon(f_b)$ is bounded. Extending a theorem of Yao [17] from the two-party to the n-party case, it follows that $C_{\det}(f_b)(m)$ is at most exponential in $C_\varepsilon(f_b)(m)$. In particular, if $C_\varepsilon(f_b)$ is bounded, then so is $C_{\det}(f_b)$. If for all m, $C_{\det}(f)(m) \leq d$, this means that for every m, there is a d-bit protocol for f on $\left(\{0,1\}^{\leq m}\right)^n$. Using König's lemma, this implies the existence of one deterministic protocol, $\mathcal{F}$, defined over $(\{0,1\}^*)^n$, which computes f and for every input exchanges at most d bits. (To prove this, consider the following tree: The nodes in level m of the tree are the protocols that compute f on $\left(\{0,1\}^{\leq m}\right)^n$ that use at most d bits of communication. A protocol in level $m+1$ is the son of a protocol in level m if they have the same communication on all inputs in $\left(\{0,1\}^{\leq m}\right)^n$. Since for every m there exists such a protocol, the tree is infinite. In addition, each protocol in level m has finitely many sons—at most the number of possibilities to map the strings of length $m+1$ into the set of d-bit communications. This implies, through the use of König's lemma, the existence of an infinite branch in the tree. This branch defines a protocol that computes f over $(\{0,1\}^*)^n$, and uses at most d bits of communication.)

Let $\{S_1, S_2, \ldots, S_k\}$ $(k \leq 2^d)$ be the set of all possible communications (in this deterministic protocol) with last message (the output) consisting of "1." We describe an $\lfloor (n-1)/2 \rfloor$-private protocol, $\mathcal{F}'$, which computes f_b. In the first step of $\mathcal{F}'$, each party P_i locally computes the set y_i as follows:

$$y_i = \{j \mid 1 \leq j \leq k, \text{ the communication } S_j \text{ is consistent with } x_i\}.$$

Let

$$f_b^*(y_1, \ldots, y_n) = \begin{cases} 1 & \text{if } \cap y_i \neq \emptyset, \\ 0 & \text{otherwise,} \end{cases}$$

that is, $f_b^*(\vec{y}) = 1$ if and only if there is a communication S_j with "output" 1, which is consistent with the inputs of all n participants. One can verify that $f_b^*(\vec{y}) = f_b(\vec{x})$ and that f_b^*'s domain is finite (every y_i is a subset of $\{1, 2, \ldots k\}$). Thus, using the protocols of [5] and [8] the value of $f_b^*(\vec{y})$ can be computed $\lfloor (n-1)/2 \rfloor$-privately, which implies that the value of $f_b(\vec{x})$ can be computed $\lfloor (n-1)/2 \rfloor$-privately. Through Lemma 3.1, the proof of the theorem is completed. □

COROLLARY 3.3. *Let* $f : (\{0,1\}^*)^n \to B$ *and* $0 < \varepsilon \leq \frac{1}{2}$. *If* $C_\varepsilon(f)$ *is bounded, then* f *is* $\lfloor (n-1)/2 \rfloor$*-private.*

COROLLARY 3.4. *Let* $f : (\{0,1\}^*)^n \to B$. *If for every* $b \in B$ *the set* $f^{-1}(b)$ *is finite, then* f *is* $\lfloor (n-1)/2 \rfloor$*-private.*

These results imply that many "natural" functions can be computed $\lfloor (n-1)/2 \rfloor$-privately. This includes functions as addition over the positive integers, maximum and minimum, and multiplication over the integers. Note that for the minimum and maximum, $f^{-1}(b)$ is infinite for every b. However, these two functions satisfy the condition of Theorem 3.2 and thus are $\lfloor (n-1)/2 \rfloor$-private.

Finally, we remark that the above protocols satisfy a stronger definition of privacy, as defined in [10].

4. Impossibility results. In this section we apply communication complexity arguments to prove the existence of functions that are not even 1-private. Specifically, we show that addition over $\mathcal{Z}$, the ring of integers, is not 1-private. The same result holds for integer gcd.

THEOREM 4.1. *Let* $0 < \varepsilon \leq \frac{1}{2}$ *and* $n \geq 3$. *Let* $SUM_n : \mathcal{Z}^n \to \mathcal{Z}$ *be defined as* $SUM_n(\vec{x}) = \sum_{i=1}^{n} x_i$. *There is no* ε*-advantage protocol which computes* SUM_n 1*-privately.*

The high-level structure of the proof is as following: Assume, towards a contradiction, that there exists an ε-advantage, 1-private protocol $\mathcal{F}$ which computes SUM_n. Privacy arguments imply that, with high probability, the number of bits communicated by $\mathcal{F}$ on all inputs $\vec{x}$ which satisfy $SUM_n(\vec{x}) = 0$ is bounded. We then show how to transform $\mathcal{F}$ into a two-argument protocol which computes the predicate "identity" using bounded communication. This contradicts a known lower bound.

LEMMA 4.2. *Let* $0 < \varepsilon \leq \frac{1}{2}$ *and* $0 < \delta \leq 1$ *be two constants. Suppose there is an* ε*-advantage,* 1*-private protocol* $\mathcal{F}$ *for computing* SUM_n. *Then there exists a constant* d *such that for every input vector* $\vec{x}$ *with* $SUM_n(\vec{x}) = 0$, *the probability that fewer than* d *bits are transmitted during the execution of* $\mathcal{F}$ *is at least* $1 - \delta$.

Proof. We first show that for every $\vec{x}$ with $SUM_n(\vec{x}) = 0$, the distribution of messages on every link (P_i, P_j) $(i < j)$ of the network, when the input is $\vec{x}$, is the same as the distribution of messages on this link, when the input is $(0, 0, \ldots, 0)$. Then, we prove the existence of a constant d as above for the input vector $(0, 0, \ldots, 0)$. Combining these two claims we get the proof of the lemma.

Let $\vec{x}$ be an input vector satisfying $SUM_n(\vec{x}) = 0$. To simplify the notation, we restrict attention to the link (P_1, P_2). Consider the following three input vectors:

$$\begin{array}{lllllll} \vec{x} & = & (x_1, & x_2, & x_3, & x_4, \ldots, & x_n), \\ \vec{x'} & = & (x_1, & 0, & x_2 + x_3, & x_4, \ldots, & x_n), \\ \vec{0} & = & (0, & 0, & 0, & 0, \ldots, & 0). \end{array}$$

By the assumptions, SUM_n is 0 for all three vectors. By 1-privacy, the distribution of messages on the link (P_1, P_2) must be the same for the two vectors $\vec{x}$ and $\vec{x'}$. Otherwise P_1 will be able to distinguish between the two. Similarly, the distribution of messages on the link (P_1, P_2) must be equal for the two vectors $\vec{x'}$ and $\vec{0}$, as otherwise P_2 will be able to distinguish between them. Thus, the distribution of messages on the link (P_1, P_2) is equal for $\vec{x}$ and $\vec{0}$.

In the second part of the proof, we use the termination condition for the input vector $\vec{0}$. It implies that for each link (P_i, P_j) there exists a constant $d_{i,j}$ such that the probability that more then $d_{i,j}$ bits are exchanged over (P_i, P_j) during the execution of the protocol on $\vec{0}$ is at most $\delta / \binom{n}{2}$.

Let $\vec{x}$ be any input vector that satisfies $SUM_n(\vec{x}) = 0$. By the first part of the proof, the probability that more than $d_{i,j}$ bits are exchanged over (P_i, P_j) during the execution of the protocol on $\vec{x}$ is also bounded by $\delta / \binom{n}{2}$. Thus, the probability that more than $d = \sum_{i<j} d_{i,j}$ bits are exchanged (over all $\binom{n}{2}$ communication lines) during the execution of $\mathcal{F}$ on $\vec{x}$ is at most δ. This completes the proof of the lemma. □

The next lemma is a known result in communication complexity. Let $ID : \{0, 1\}^m \times \{0, 1\}^m \to \{0, 1\}$ be the following function:

$$ID(a, b) = \begin{cases} 1 & \text{if } a = b, \\ 0 & \text{otherwise.} \end{cases}$$

LEMMA 4.3 (Yao [17]). *For every constant* $0 < \varepsilon \leq \frac{1}{2}$, $C_\varepsilon(ID)(m) = \Theta(\log m)$.

Proof of Theorem 4.1. Assume, towards a contradiction, that there exists a protocol $\mathcal{F}$ which computes SUM_n 1-privately with ε-advantage. Let $\delta = \frac{\varepsilon}{2}$. By using Lemma 4.2 we

see that there exists a constant d such that for every input vector $\vec{x}$ that satisfies $SUM_n(\vec{x}) = 0$, the probability that more than d bits are transmitted during the execution of $\mathcal{F}$ is bounded by δ. We use this property of $\mathcal{F}$ to construct a randomized protocol $\mathcal{A}$ to compute the ID function over $\{0,1\}^m \times \{0,1\}^m$, with $\varepsilon/2$ - advantage, which uses only a constant $(d+1)$ number of bits. This will be a contradiction to Lemma 4.3.

The protocol $\mathcal{A}$ works as following: On input $(a,b) \in \{0,1\}^m \times \{0,1\}^m$, the two parties (denoted by P_1^* and P_2^*) will simulate the protocol $\mathcal{F}$:

- P_1^* will simulate P_1 with $x_1 = a$ (the integer in the range $[0, 2^m - 1]$ whose binary representation is the string a).
- P_2^* will simulate $P_2, P_3, \ldots, P_n$ with $x_2 = -b$ and $x_3 = x_4 = \ldots = x_n = 0$.

The parties P_1^* and P_2^* will simulate $\mathcal{F}$ until it either terminates, or d bits are transmitted. If $\mathcal{F}$ terminates with final value 0, the parties output $ID(a,b) = 1$ (specifically, they claim that $a = b$). If $\mathcal{F}$ terminates with value $\neq 0$, or does not terminate after exchanging d bits, then the parties output $ID(a,b) = 0$ (that is, they claim that $a \neq b$).

Clearly, the protocol $\mathcal{A}$ transmits at most $d+1$ bits. What we now show is that $\mathcal{A}$ computes the value of the ID function over $\{0,1\}^m \times \{0,1\}^m$ with probability $\geq \frac{1}{2} + \frac{\varepsilon}{2}$. If $a = b$, then the constructed $\vec{x}$ satisfies $SUM_n(\vec{x}) = 0$. Through Lemma 4.2, the probability that more than d bits are exchanged on such input is at most δ. The other possible source of error is an incorrect output of $\mathcal{F}$. The probability that this happens is no greater than $\frac{1}{2} - \varepsilon$. Thus, the overall error probability of $\mathcal{A}$ on (a,b) is no greater than $\frac{1}{2} - \varepsilon + \delta$. By the choice of δ this is equal to $\frac{1}{2} - \frac{\varepsilon}{2}$. If $a \neq b$ then $\vec{x}$ satisfies $SUM_n(\vec{x}) \neq 0$. If more than d bits are transmitted, then $\mathcal{A}$ outputs 0, which is the correct value of ID in this case. As the probability of error in $\mathcal{F}$ is bounded by $\frac{1}{2} - \varepsilon$, then clearly this is so in case that the execution terminates with at most d communicated bits. Thus, on such (a,b) the protocol $\mathcal{A}$ computes the correct value of $ID(a,b)$ with probability at least $\frac{1}{2} + \varepsilon$. This completes the proof. □

COROLLARY 4.4. *Multiplication over the rationales is not* 1*-private.*

Proof. Addition over the integers can be reduced to multiplication in the cyclic group $\{2^i \mid i \in \mathcal{Z}\}$ by mapping the integer i to the rational number 2^i. Thus a 1-private protocol for multiplication over the rationales would translate into a 1-private protocol for addition over the integers, which contradicts Theorem 4.1. □

The technique relating communication complexity and privacy is applicable for various other functions. For example, we have the following theorem.

THEOREM 4.5. *Let* $0 < \varepsilon \leq \frac{1}{2}$ *and* $n \geq 2$. *Let* $\mathcal{N}$ *denote the set of natural numbers* $\{1, 2, \ldots\}$, *and* $GCD_n : \mathcal{N}^n \to \mathcal{N}$ *be defined as the greatest common divisor of the* n *arguments. There is no* ε*-advantage protocol which computes* GCD_n 1*-privately.*

Proof. The proof of this theorem is very similar to the proof for Theorem 4.1. We sketch the differences: the analogue of Lemma 4.2 will be proved for input vectors satisfying $GCD_n(\vec{x}) = 1$. In the proof of the analogous lemma we use the following triples of n-tuples:

$$\begin{array}{rcl} \vec{x} & = & (\ x_1,\ x_2,\ x_3,\ x_4, \ldots,\ x_n\ ), \\ \vec{x'} & = & (\ x_1,\ 1,\ x_3,\ x_4, \ldots,\ x_n\ ), \\ \vec{1} & = & (\ 1,\ 1,\ 1,\ 1, \ldots,\ 1\ ). \end{array}$$

Now, let $PRIMES = \{q_1, q_2, q_3, \ldots\}$ denote the sequence of prime numbers. In the proof of the theorem we will construct a protocol for ID as follows: On input (a,b) (two natural numbers ≥ 1)

- P_1^* will simulate P_1 with $x_1 = q_a$.
- P_2^* will simulate $P_2, P_3, \ldots, P_n$ with $x_2 = x_3 = \ldots = x_n = q_b$.

Clearly, $GCD_n(\vec{x}) = 1$ if and only if $a \neq b$. Thus the original argument holds here as well. □

5. Characterization for Boolean functions. In this section we characterize the privacy of any Boolean function f in terms of its communication complexity. Corollary 3.3 implies that any Boolean function f with bounded communication complexity is $\lfloor (n-1)/2 \rfloor$-private. The main theorem in this section states the reverse implication. More precisely, a Boolean function whose communication complexity is unbounded is not even 1-private. We set the stage for the theorem by introducing some definitions and claims.

Let $A \subseteq \{0,1\}^* \times \{0,1\}^*$. Define a relation $\sim_A$ on $A \times A$ by $(x_1, y_1) \sim_A (x_2, y_2)$ if $x_1 = x_2$ or $y_1 = y_2$. The equivalence relation $\equiv_A$ is defined as the transitive closure of $\sim_A$. $Class(A)$ denotes the number of equivalence classes of $\equiv_A$ (it can be infinite). For any set $S \subseteq \{0,1\}^* \times \{0,1\}^*$, define S_1 and S_2 as the projections of S on the first and second coordinates, respectively. We now present two simple properties of the equivalence classes of the relation previously defined.

CLAIM 1. Let $A \subseteq \{0,1\}^* \times \{0,1\}^*$, and let K and L be two disjoint (nonempty) equivalence classes of $\equiv_A$. Then $K_1 \cap L_1 = K_2 \cap L_2 = \emptyset$.

Proof. The proof follows directly from the definition of the equivalence relation $\equiv_A$. □

CLAIM 2. Let $A \subseteq \{0,1\}^* \times \{0,1\}^*$, and let K and L be two disjoint (nonempty) equivalence classes of $\equiv_A$. If $(x, y) \in (K_1 \times L_2) \cup (L_1 \times K_2)$, then $(x, y) \notin A$.

Proof. Assume, without loss of generality, that $(x, y) \in (K_1 \times L_2)$. This implies that there exist y' such that $(x, y') \in K$ and x' such that $(x', y) \in L$. Suppose that $(x, y) \in A$. By the definition of $\sim_A$ these pairs satisfy $(x, y') \sim_A (x, y) \sim_A (x', y)$, contradicting the disjointness of the two $\equiv_A$ equivalence classes K and L. □

The next lemma states that if A and B cover $\{0,1\}^* \times \{0,1\}^*$ (A and B need not be disjoint), then either $Class(A) \leq 2$ or $Class(B) \leq 2$. Furthermore, if $Class(B) \geq 3$ then $Class(A) = 1$. Notice that it is possible to have $Class(A) = Class(B) = 2$, e.g., by taking $A = \{(x, y) | x \equiv y \pmod 2\}$ and $B = \{(x, y) | x \not\equiv y \pmod 2\}$.

LEMMA 5.1. *Suppose that* $\{0,1\}^* \times \{0,1\}^* \subseteq A \cup B$. *If* $Class(B) \geq 3$, *then* $Class(A) = 1$.

Proof.[2] Let C, D, E be three distinct equivalence classes of $\equiv_B$. Let (x_1, y_1), (x_2, y_2), and (x_3, y_3) be arbitrary elements of C, D and E, respectively (see Fig. 1). By Claim 1, $x_i \neq x_j$ and $y_i \neq y_j$ for every $i \neq j$ $(1 \leq i, j \leq 3)$. Consider the element (x_1, y_2). By Claim 2, $(x_1, y_2) \in A$. Similarly, $(x_i, y_j) \in A$ for every $i \neq j$ $(1 \leq i, j \leq 3)$. Furthermore, by Claim 1, they all belong to the same equivalence class of $\equiv_A$. Denote this class by F.

	y_1	y_2	y_3	y_4
x_1	C	F	F	-
x_2	F	D	F	-
x_3	F	F	E	-
x_4	?	?	-	G

FIG. 1

Assume, towards a contradiction, that $\equiv_A$ has another equivalence class G and let $(x_4, y_4) \in G$. By Claim 1, $x_4 \neq x_j$ and $y_4 \neq y_j$ for every $1 \leq j \leq 3$. Consider the element (x_4, y_1). By Claim 2, $(x_4, y_1) \notin A$. Therefore, $(x_4, y_1) \in B$ and by Claim 1 it must belong to C. By a similar argument, (x_4, y_2) must belong to D. This implies that $C_1 \cap D_1 \neq \emptyset$, which contradicts Claim 1. □

[2]This proof simplifies our original proof, which appeared in the early version of this paper. It is due to Freddy Bruckstein and Alon Orlitsky, and it appears here with their kind permission.

THEOREM 5.2. *Let* $f : (\{0,1\}^*)^n \to \{0,1\}$ *be* 1*-private. Then the communication complexity of* f *is bounded.*

Proof. Let $f : (\{0,1\}^*)^n \to \{0,1\}$, and $\mathcal{F}$ be a 1-private protocol which computes f with ε-advantage. We show that $\mathcal{F}$ can be modified to a communication protocol which computes f with $\frac{\varepsilon}{2}$-advantage while exchanging only a bounded number of bits between every pair of parties (where the bound does not depend on the input).

In order to carry out this transformation, it would be desirable to be in a situation similar to that occurring in SUM_n, where every $\vec{z} \in f^{-1}(0)$ has message distribution similar to some *fixed* input ($\vec{0}$ for SUM_n). While this desired situation may not occur, something weaker, but sufficient for our purposes, does. For every link (P_i, P_j), there is a value $v \in \{0,1\}$ and there are at most two *fixed* inputs $\vec{x}(i,j), \vec{w}(i,j) \in (\{0,1\}^*)^n$ so that for every $\vec{z} \in f^{-1}(v)$, the distribution of messages exchanged on the link (P_i, P_j), given input $\vec{z}$, is *identical* to the same distribution either given input $\vec{x}(i,j)$ or given input $\vec{w}(i,j)$.

Let $1 \le i < j \le n$. Define the set $A_{i,j}$ (resp., $B_{i,j}$) by $(x_i, x_j) \in A_{i,j}$ if there exists an n-tuple $\vec{x} = (x_1, x_2, \ldots, x_n)$ extending (x_i, x_j) such that $f(\vec{x}) = 0$ (resp., $f(\vec{x}) = 1$). Notice that $A_{i,j} \cup B_{i,j}$ covers $\{0,1\}^* \times \{0,1\}^*$. We now relate the 1-privacy of f to the sets $A_{i,j}$, $B_{i,j}$.

Suppose $f(\vec{x}) = f(\vec{w}) = 0$ and $(x_i, x_j) \equiv_{A_{i,j}} (w_i, w_j)$. Any 1-private protocol which computes f induces identical distribution of messages exchanged between P_i and P_j, given the input $\vec{x}$ and the input $\vec{w}$. (Similarly for $f(\vec{x}) = f(\vec{w}) = 1$ and $(x_i, x_j) \equiv_{B_{i,j}} (w_i, w_j)$.) The proof of this claim is immediate from the definitions of the sets $A_{i,j}$, $B_{i,j}$, the relations $\sim_{A_{i,j}}$, $\sim_{B_{i,j}}$, $\equiv_{A_{i,j}}$ and $\equiv_{B_{i,j}}$, and 1-privacy.

By Lemma 5.1, for each (i,j), at least one of the sets $A_{i,j}$, $B_{i,j}$ has no more than two equivalence classes in the corresponding $\equiv$ equivalence relation. Without loss of generality, $A_{i,j}$ has at most two equivalence classes, and let (x_i, x_j) and (w_i, w_j) be fixed representatives of the two classes. (Notice that for a different pair i', j', the set $B_{i',j'}$ can be the set with $Class(\cdot) \le 2$.) Let $\vec{x}(i,j)$ and $\vec{w}(i,j)$ denote two inputs in $f^{-1}(0)$ whose projections on the i and j coordinates yield the above representatives (take $\vec{x}(i,j) = \vec{w}(i,j)$ if there is only one equivalence class).

Let $\mathcal{F}$ be a 1-private protocol which computes f with ε-advantage ($0 < \varepsilon \le \frac{1}{2}$). Consider runs of $\mathcal{F}$ on the $2 \cdot \binom{n}{2}$ inputs $\vec{x}(1,2), \vec{w}(1,2), \ldots, \vec{x}(n-1,n), \vec{w}(n-1,n)$. Let d be the minimal integer such that the probability that P_i and P_j exchange more than d bits, given that the input is either $\vec{x}(i,j)$ or $\vec{w}(i,j)$, is less than ε/n^2 (for all (i,j) pairs).

We modify the protocol $\mathcal{F}$ as follows: All parties run $\mathcal{F}$ until it either terminates, or some pair attempts to exchange more than d bits. Suppose, without loss of generality, that the first such pair is $(1,2)$, and further that $\vec{x}(1,2), \vec{w}(1,2) \in A_{1,2}$. In this case, the pair aborts the execution of $\mathcal{F}$, and announces that the value of f is 1. (In case of conflicting announcements in the same round, the one resulting from the minimal pair in lexicographical order is adopted by all parties.)

The modified protocol exchanges at most $d \cdot \binom{n}{2}$ bits on all inputs. We argue that it computes the correct answer with probability at least $\frac{1}{2} + \frac{\varepsilon}{2}$. Suppose, without loss of generality, that $f(\vec{z}) = 0$. For all pairs i, j with representatives $\vec{x}(i,j), \vec{w}(i,j) \in A_{i,j}$, the messages exchanged between P_i and P_j on input $\vec{z}$ are distributed identically to one of the two representatives, and thus the probability that more than d bits will be exchanged in $\mathcal{F}$ is bounded above by ε/n^2. Thus the probability that some i, j with representatives in $A_{i,j}$ will abort the execution of $\mathcal{F}$ and announce "1" as the final outcome is smaller then $\varepsilon/2$. Any pair i, j with $\vec{x}(i,j), \vec{w}(i,j) \in B_{i,j}$ cannot announce "1" as the final outcome by aborting $\mathcal{F}$'s execution. The other source of errors stems from errors in $\mathcal{F}$'s final outcome. The probability that this occurs is smaller than $\frac{1}{2} - \varepsilon$. Thus the overall probability of error in the modified protocol is smaller than $\frac{1}{2} - \frac{\varepsilon}{2}$. □

By combining Theorem 5.2 and Corollary 3.3, we get the next corollary.

COROLLARY 5.3. *A Boolean function* $f : (\{0,1\}^*)^n \to \{0,1\}$ *is* 1*-private if and only if it has bounded communication complexity. If* f *is* 1*-private, then it is* $\lfloor (n-1)/2 \rfloor$*-private.*

Together with the characterization of $\lceil n/2 \rceil$-private Boolean functions of [10], we conclude with the following theorem.

THEOREM 5.4. *Any Boolean function* $f : (\{0,1\}^*)^n \to \{0,1\}$ *is either*

- n*-private.*
- $\lfloor (n-1)/2 \rfloor$*-private private but not* $\lceil n/2 \rceil$*-private.*
- *not* 1*-private.*

6. Nonterminating protocols. So far, we considered protocols that are private, compute the correct value (with probability at least $\frac{1}{2} + \varepsilon$), and terminate (with probability 1). In this section we show that if we abandon the termination condition (and the requirement that a distinguished "last message" is sent), and require instead only "correctness in the limit," then private computations which were not possible before become possible. We exemplify the model via two examples. Consider the following nonterminating protocol for computing SUM_n:

For $k = 1, 2, 3, \ldots,$

- each party P_i locally computes $y_i \in [-k, k]$ such that $y_i \equiv x_i \pmod{2k+1}$.
- The parties compute n-privately, using the protocol of [6], s_k such that $s_k \equiv \sum_{i=1}^{n} y_i \pmod{2k+1}$ and $s_k \in [-k, k]$. The value s_k is the outcome of the kth round.

It is easy to see that no coalition (of any size $t \leq n$) gets any additional information from the execution of this protocol. After a *finite* number of iterations (which depends on the input vector) the output s_k stabilizes on the correct value. Thus, SUM_n is n-private in the nonterminating model.

The next example shows that we can demand even more. Consider the following nonterminating protocol for computing GCD_n:

For $k = 1, 2, 3, \ldots,$

- each party P_i computes $y_i = 1$ if $k \mid x_i$ and $y_i = 0$ otherwise.
- The parties compute, $\lfloor (n-1)/2 \rfloor$-privately, the AND of the y_i's. This bit indicates whether k divides all the x_i's or not. The output of the kth round is the maximal $g \leq k$ that divides all the x_i's.

The protocol computes the function GCD_n, and it is $\lfloor (n-1)/2 \rfloor$-private in the nonterminating model. In the protocol for SUM_n, no participant knows when the result stabilizes. Here, however, the gcd can never exceed x_i. Thus after x_i many iterations, party P_i knows that he knows the final outcome. After $\max_{i=1}^{n} x_i$ iterations, all the parties know that they know the final outcome. However, the protocol must continue, since the parties do not know whether the other parties know that the result is already final. In other words, the privacy requirements prevent the parties from achieving *common knowledge* [14] on the fact that the result is already final, and an infinite execution is enforced.

7. Concluding remarks. In case that a function f is not 1-private, it can still be computed by first revealing the lengths $|x_i|$ of all participants inputs, and then by using the protocols for the finite case of [8] and [5]. However, $f(\vec{x})$ can always be computed while revealing only smaller amount of additional information (for a formal definition see [3] and [4])—the *maximal* input length, i.e., $\max_{i=1}^{n} |x_i|$—in the following two steps:

- Compute $\lfloor (n-1)/2 \rfloor$-privately the value $m = \max_{i=1}^{n} |x_i|$.
- Compute $\lfloor (n-1)/2 \rfloor$-privately the value of f over the *finite* domain of strings with length at most m.

For functions with low circuit complexity, the second step can be performed efficiently using the protocols described in [5], [8], and [1]. The first step can be performed by using the protocol described in the proof of Theorem 3.2 (§3): In the ℓth round, party P_i sets $y_i = 1$ if $|x_i| \leq \ell$ and $y_i = 0$ otherwise. Then, the parties compute $\lfloor (n-1)/2 \rfloor$-privately the AND of the y_i's and stop with output ℓ if the AND is 1. This step is therefore polynomial time in the length of the input $\vec{x}$. It is possible to substantially reduce the number of iterations in this part while preserving the polynomial complexity, by checking in the ℓth round whether $\max_{i=1}^{n} |x_i| \leq 2^{\ell}$. Thus if the maximum is m, this fact is found in $O(\log m)$ rounds (instead of $O(m)$ rounds).

A possible direction for extending Theorem 5.2 to functions with non-Boolean range might be to generalize Lemma 5.1 to the coloring of the two-dimensional plane by more than two colors. However, such generalizations of the lemma are not true: There are three colorings of the plane (say by colors 0,1,2), where $Class(0) = Class(1) = Class(2) = \infty$. Such example was constructed by Nati Linial, and is included, with his kind permission, in the Appendix. It remains open whether the corresponding generalization of Theorem 5.2 is true.

One of the interesting questions in the area of private computations has been to investigate the structure of the *privacy hierarchy*. Recently, we have resolved this question [12] for n-argument functions which are defined over finite domains. We have shown that for finite domains, the privacy hierarchy has exactly $\lfloor n/2 \rfloor$ levels. For any $\lceil n/2 \rceil \leq t \leq n-2$, an n–argument function which is t-private but not $t+1$-private was constructed. The major tool used in this proof was to partition the n inputs into two sets of appropriate sizes and apply the criteria for privacy of two argument functions. This approach is not useful in determining t-privacy for t which is at most $\lfloor (n-1)/2 \rfloor$, because one of the two sizes will be larger than t. Indeed, the results in [12] leave open the structure of the privacy hierarchy for functions defined over countable domains.

The communication complexity techniques used in the present work are of a very different nature than the partition techniques which we have just mentioned. They enable us to show that the privacy hierarchy contains a new, additional level (0-privacy). We believe that these techniques will also play an important role in future research towards resolving the exact structure of the privacy hierarchy over countable domains.

Appendix. In this appendix we describe an example, due to Nati Linial, of a function $f : \mathcal{N} \times \mathcal{N} \to \{0, 1, 2\}$ with the property that $Class(0) = Class(1) = Class(2) = \infty$. This example implies that Lemma 5.1 cannot be extended to non-Boolean functions. The function f is defined as follows (see Fig. 2):

$$f(x, y) = \begin{cases} x \bmod 3 & \text{if } x = y, \\ x + 1 \bmod 3 & \text{if } x \neq y \wedge (x \equiv y) \bmod 3, \\ 2(x + y) \bmod 3 & \text{otherwise.} \end{cases}$$

It can be verified that every diagonal point (x, x) forms a singleton equivalence class. Since for each value in $\{0, 1, 2\}$ there are infinitely many preimages of the form (x, x), it follows that $Class(0) = Class(1) = Class(2) = \infty$.

Acknowledgments. We are grateful to Shai Ben-David, Josh Benaloh, and Charlie Rackoff for very helpful discussions and contributions to this work. Thanks to Freddy Bruckstein and Alon Orlitsky for simplifying the proof of Lemma 5.1 and to Nati Linial for his three-coloring example. We would also like to thank Oded Goldreich, who has the good quality of commenting quickly on early drafts.

	0	3	6	⋯	1	4	7	⋯	2	5	8	⋯
0	0	1	1	⋯	2	2	2	⋯	1	1	1	⋯
3	1	0	1	⋯	2	2	2	⋯	1	1	1	⋯
6	1	1	0	⋯	2	2	2	⋯	1	1	1	⋯
⋮	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋱
1	2	2	2	⋯	1	2	2	⋯	0	0	0	⋯
4	2	2	2	⋯	2	1	2	⋯	0	0	0	⋯
7	2	2	2	⋯	2	2	1	⋯	0	0	0	⋯
⋮	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋱
2	1	1	1	⋯	0	0	0	⋯	2	0	0	⋯
5	1	1	1	⋯	0	0	0	⋯	0	2	0	⋯
8	1	1	1	⋯	0	0	0	⋯	0	0	2	⋯
⋮	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋱

FIG. 2

REFERENCES

[1] J. BAR-ILAN AND D. BEAVER, *Noncryptographic fault-tolerant computing in a constant number of rounds*, Proc. of 8th ACM Symposium on Principles of Distributed Computing, 1989, ACM Press, New York, pp. 201–209.

[2] D. BEAVER, *Perfect privacy for two party protocols*, Tech. Rep. TR-11-89, Harvard University, Cambridge, MA, 1989.

[3] R. BAR-YEHUDA, B. CHOR, AND E. KUSHILEVITZ, *Privacy, additional information, and communication*, Proc. of 5th IEEE Structure in Complexity Theory, IEEE Press, July 1990, pp. 55–65.

[4] R. BAR-YEHUDA, B. CHOR, E. KUSHILEVITZ, AND A. ORLITSKY, *Privacy, additional information, and communication*, IEEE Trans. Inform. Theory, 39 (1993), pp. 1930–1943.

[5] M. BEN-OR, S. GOLDWASSER, AND A. WIGDERSON, *Completeness theorems for non-cryptographic fault-tolerant distributed computation*, Proc. of 20th ACM Symposium on Theory of Computing, ACM Press, New York, 1988, pp. 1–10.

[6] J. D. BENALOH (COHEN), *Secret sharing homomorphisms: Keeping shares of a secret secret*, Advances in Cryptography - Crypto86 (proceedings), A. M. Odlyzko, ed., Lecture Notes in Comput. Sci., 263, pp. 251–260, Springer-Verlag, New York, 1987.

[7] G. R. BLAKLEY AND L. SWANSON, *Security proofs for information protection systems*, Proc. of IEEE Symposium on Security and Privacy, IEEE Press, 1981, pp. 75–88.

[8] D. CHAUM, C. CRÉPEAU, AND I. DAMGÅRD, *Multiparty unconditionally secure protocols*, Proc. of 20th ACM Symposium on Theory of Computing, ACM Press, New York, 1988, pp. 11–19.

[9] A. K. CHANDRA, M. L. FURST, AND R. J. LIPTON, *Multi-party protocols*, Proc. of 15th ACM Symposium on Theory of Computing, ACM Press, New York, 1983, pp. 94–99.

[10] B. CHOR AND E. KUSHILEVITZ, *A zero-one law for Boolean privacy*, SIAM J. Disc. Math., 4 (1991), pp. 36–47.

[11] ———, *Secret sharing over infinite domains*, J. Cryptology, 6 (1993), pp. 87–96.

[12] B. CHOR, M. GERÉB-GRAUS, AND E. KUSHILEVITZ, *On the structure of the privacy hierarchy*, J. Cryptology, 7 (1994), pp. 53–60.

[13] O. GOLDREICH, S. MICALI, AND A. WIGDERSON, *How to play any mental game*, Proc. of 19th ACM Symposium on Theory of Computing, ACM Press, New York, 1987, pp. 218–229.

[14] J. Y. HALPERN AND Y. MOSES, *Knowledge and common knowledge in distributed environment*, J. Assoc. Comput. Mach., 37 (1990), pp. 549–587.

[15] E. KUSHILEVITZ, *Privacy and communication complexity*, SIAM J. Disc. Math., 5 (1992), pp. 273–284.

[16] P. TIWARI, *Lower bounds on communication complexity in distributed computer networks*, J. Assoc. Comput. Mach., 34 (1987), pp. 921–938.

[17] A. C. YAO, *Some complexity questions related to distributive computing*, Proc. of 11th ACM Symposium on Theory of Computing, ACM Press, New York, 1979, pp. 209–213.

[18] ———, *Protocols for secure computations*, Proc. of 23th Annual IEEE Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1982, pp. 160–164.

SIAM J. COMPUT.
Vol. 24, No. 2, pp. 387–397, April 1995

014

SPARSE POLYNOMIAL INTERPOLATION IN NONSTANDARD BASES*

Y. N. LAKSHMAN† AND B. DAVID SAUNDERS‡

Abstract. In this paper, we consider the problem of interpolating univariate polynomials over a field of characteristic zeros that are sparse in (a) the Pochhammer basis, or (b) the Chebyshev basis. The polynomials are assumed to be given by black boxes, i.e., one can obtain the value of a polynomial at any point by querying its black box. We describe efficient new algorithms for these problems. Our algorithms may be regarded as generalizations of Ben-Or and Tiwari's (1988) algorithm (based on the BCH decoding algorithm) for interpolating polynomials that are sparse in the standard basis. The arithmetic complexity of the algorithms is $O(t^2 + t \log d)$, which is also the complexity of the univariate version of the Ben-Or and Tiwari algorithm. That algorithm and those presented here also share the requirement of $2t$ evaluation points.

Key words. polynomial interpolation, sparsity, Chebyshev polynomial, Pochhamer basis, BCH codes

AMS subject classifications. 68Q40, 12Y05, 41A05

Introduction. In this paper, we consider the problem of efficiently interpolating polynomials given by black boxes that have sparse representations in various bases. Usually, one considers a polynomial $f(x) = \sum_{i=0}^{n} a_i x^i$ as t-sparse if at most t of the coefficients a_i are non-zero. The problem of interpolating a sparse polynomial from a list of values at specific points, or from values given by a black box that evaluates the polynomial is of great importance in computational algebra. Sparse interpolation has received significant attention in several recent papers. (Ben-Or and Tiwari (1988), Clausen, Dress, Grabmeier, and Karpinski (1988), Kaltofen and Lakshman (1988), Grigoriev and Karpinski (1987), Grigoriev, Karpinski, and Singer (1990), (1991a), (1994), Borodin and Tiwari (1990), Zippel (1990).) In particular, the problem of interpolating a t-sparse univariate polynomial given a black box for evaluating the polynomial can be efficiently solved using Ben-Or and Tiwari's adaptation of the BCH decoding algorithm.

Sparse interpolation is well recognized as a very useful tool for controlling intermediate expression swell in computer algebra (Zippel (1990), Kaltofen and Trager (1990)). Sparse polynomials and rational functions can be evaluated quickly and that makes them attractive to several applications and an interesting line of research is to try to infer properties of sparse polynomials (such as divisibility, existence of nontrivial greatest common divisor, the existence of real roots, etc.) from their values at a small number points (see Grigoriev, Karpinski, and Odlyzko (1992)). Traditionally, "sparse polynomial (or rational function)" is taken to mean a polynomial (or rational function) with a "few terms" where the "terms" are power products of the variables involved. One can reasonably ask for sparse representations for polynomials in other bases such as the Chebyshev polynomials or the shifted power basis $1, x - \alpha, (x - \alpha)^2, \ldots$. In this paper, we consider two classes of polynomials—those which are sparse in the Chebyshev basis and those which are sparse in the Pochhammer basis. A polynomial $f(x) = \sum_{i=0}^{n} a_i T_i(x)$ is t-sparse in the Chebyshev basis if at most t of the coefficients a_i are nonzero, where $T_i(x)$ is the ith Chebyshev polynomial. Sparse Pochhammer polynomials are defined similarly. We provide algorithms for interpolating such polynomials efficiently, given a black box that evaluates the polynomial. The algorithms require as input an upper bound t on the number of nonzero terms in the interpolating polynomial. The arithmetic complexity of the algorithms is $O(t^2 + t \log d)$, which is also the complexity of the

*Received by the editors October 1, 1992; accepted for publication (in revised form) November 11, 1993.

†Department of Mathematics and Computer Science, Drexel University, Philadelphia, Pennsylvania 19104 (lakshman@mcs.drexel.edu). The work of this author was supported in part by NSF grant CCR–9203062.

‡Department of Computer and Information Sciences, University of Delaware, Newark, Delaware 19716 (saunders@udel.edu). The work of this author was supported in part by NSF grant CCR-9123666.

univariate version of the Ben-Or and Tiwari algorithm. That algorithm and those presented here also share the requirement of $2t$ evaluation points. However, when one considers the bit complexity (when the ground field is $\mathbf{Q}$), the Ben-Or and Tiwari algorithm and our sparse Chebyshev algorithm involve intermediate values of bit size $O(td)$, whereas in the sparse Pochhammer algorithm intermediate values grow to a bit length of $O(d \log(t))$.

Several recent results concerning sparse interpolation are similar in spirit to the BCH-decoding algorithm. In Dress and Grabmeier (1991), it is shown that many of the sparse interpolation algorithms can be formulated and proven correct in the context of t-sparse sums of abelian monoids. A somewhat different framework in Grigoriev, Karpinski, and Singer (1991a) in terms of t-sparse sums of eigenfunctions of operators is shown to provide several generalizations of the sparse interpolation algorithm of Ben-Or and Tiwari. In fact, an efficient algorithm for interpolating polynomials sparse in the Pochhammer basis follows from an observation in Grigoriev, Karpinski, and Singer (1991a) which points out that t-sparse sums of eigenfunctions of the operator $x\Delta(f) = x(f(x) - f(x-1))$ correspond to polynomials that are t-sparse in the Pochhammer basis. We use the Pochhammer case as a motivating example for our discussion of the Chebyshev case.

The problem of efficiently interpolating polynomials that are sparse in the Chebyshev basis was stated as an open problem in Borodin and Tiwari (1990). Our algorithm provides a solution and it is another generalization of the algorithm of Ben-Or and Tiwari. It uses properties shared by the standard power basis and the Chebyshev polynomials and appears to be different from the generalizations presented in Dress and Grabmeier (1991), Grigoriev, Karpinski, and Singer (1991a). While the general structure of our algorithm appears similar to the algorithms falling into the Dress–Grabmeier framework, the details are quite different. In particular, the centerpiece of the algorithms in the Dress–Grabmeier framework is a certain Toeplitz matrix where, as in our algorithm, a similar role is played by a matrix that is *the sum of a Toeplitz matrix and a Hankel matrix*. It would be interesting to reconcile our algorithm with the Dress–Grabmeier or the Grigoriev–Karpinski–Singer framework.

The rest of the paper is organized as follows. In §1, we motivate our discussion by briefly stating the algorithm for interpolating polynomials that are sparse in the Pochhammer basis. In §2, we describe our algorithm for interpolating polynomials sparse in the Chebyshev basis. Section 3 provides an analysis of the complexity of the algorithm, which is followed by some additional remarks in the last section.

1. Sparse interpolation in the Pochhammer basis. The Pochhammer symbol $x^{\overline{n}}$ denotes the rising factorial power

$$x(x+1)\dots(x+n-1)$$

for any integer $n \geq 0$. It is easily shown that $x^{\overline{i}}$, $i = 0, 1, 2, \dots$ is a $\mathbf{Q}$ basis for the polynomial ring $\mathbf{Q}[x]$. A polynomial $f(x)$ is t-sparse in the Pochhammer basis (of rising powers) if and only if $\exists f_i \in \mathbf{Q}$, $e_i \in \mathcal{Z}_+$, $i = 1, \dots, t$, such that

$$f(x) = \sum_{k=1}^{t} f_k x^{\overline{e_k}}.$$

We assume that $e_1 > e_2 > \cdots > e_t$. Suppose we are given a black box that returns the value of $f(x)$ at any $x \in \mathbf{Q}$ and a bound t on the number of nonzero terms in the representation of $f(x)$ in the Pochhammer basis. Then, we can interpolate $f(x)$ from its values at $x = 1, 2, \dots, 2t$. The algorithm follows from an observation in Grigoriev, Karpinski, and Singer (1991a) which places this problem in the framework of Dress and Grabmeier. It is an elementary fact that the finite difference operator behaves like the derivative on the Pochhammer symbol. More

precisely, let

$$\Delta(f(x)) = f(x+1) - f(x).$$

We have

$$\begin{aligned}\Delta(x^{\overline{n}}) &= (x+1)^{\overline{n}} - x^{\overline{n}} \\ &= n(x+1)^{\overline{n-1}}.\end{aligned}$$

Let

$$f^{(i)}(x) = \sum_{k=1}^{t} e_k^i f_k x^{\overline{e_k}} \text{ for } i = 0, 1, \ldots.$$

Note that

$$x\Delta(f^{(i)}(x)) = f^{(i+1)}(x).$$

We can compute $f^{(i)}(a)$ for $i = 0, 1, \ldots, 2t-1$ by repeated applications of the above recurrence from the values $f(a+i)$, $i = 0, 1, \ldots, 2t-1$.

Consider the polynomial $\Psi(z)$ of degree t whose roots are the e_i, $i = 1, \ldots, t$, i.e.,

$$\Psi(z) = \prod_{i=1}^{t}(z - e_i).$$

Let

$$\Psi(z) = z^t + \psi_{t-1}z^{t-1} + \cdots + \psi_1 z + \psi_0.$$

The $f^{(i)}$ and ψ_i satisfy the following linear relations. Assume that $\psi_t = 1$.

LEMMA 1. *For* $j = 0, \ldots, t-1$, $\sum_{i=0}^{t} \psi_i f^{(i+j)}(a) = 0$.

Proof. This is a special case of Theorem 1 in Dress and Grabmeier (1991). (See page 62 in that paper.) □

Furthermore, Theorem 1 in Dress and Grabmeier (1991) shows that if t is the number of nonzero terms in the representation of $f(x)$ in the Pochhammer basis, then the $t \times t$ matrix $\mathcal{F}$ with $\mathcal{F}_{i,j} = f^{(i+j-2)}(a)$ is nonsingular for any $a > 0$. By using this, one can compute the polynomial $\Psi(z)$ from $f(a), f(a+1), \ldots, f(a+2t-1)$. The roots of $\Psi(z)$ are the "Pochhammer exponents" of $f(x)$. Knowing these one can easily compute the coefficients f_k. We omit details to avoid repetition but state the algorithm for the sake of completeness.

Algorithm Sparse-Pochhammer-Interpolation

Input: A black box for evaluating a polynomial $f(x)$. $f(x)$ is known to be t-sparse in the Pochhammer basis.

Output: A list of pairs (f_i, e_i), $i = 1, \ldots, t$ such that $f(x) = \sum_{i=1}^{t} f_i x^{\overline{e_i}}$.

1. *Query the black box to obtain the values of* $f(x)$ *at* $x = a, a+1, \ldots, a+2t-1$ *for some* $a \neq 0$. *Compute* $f^{(i)}(a)$ *for* $i = 0, 1, \ldots, 2t-1$ *from* $f(a+i)$, $i = 0, 1, \ldots, 2t-1$.
2. *Solve the system of linear equations* $\mathcal{F}\vec{\psi} = -\vec{\mathsf{f}}$. *If*

$$\vec{\psi} = (\ \psi_0 \quad \psi_1 \quad \ldots \quad \psi_{t-1}\ )^T,$$

then the auxiliary polynomial $\Psi(z)$ *is given by*

$$\Psi(z) = z^t + \psi_{t-1}z^{t-1} + \cdots + \psi_1 z + \phi_0.$$

3. *Find the integer roots of* $\Psi(z)$. *The roots are the exponents of* $f(x)$, *i.e.,* $e_1, e_2, \ldots, e_t$.
4. *Solve the linear system of equations* $\mathcal{V}\vec{\mathsf{f}} = \vec{\mathsf{F}}$ *to obtain* $f_1 a^{\overline{e_1}}, \ldots, f_t a^{\overline{e_t}}$. *Since we know the* e_i *and* a, *the coefficients* f_i *are easily determined. Output the list of pairs* $[(f_1, e_1), \ldots, (f_t, e_t)]$.

Analysis of the Sparse-Pochhammer-Interpolation algorithm. While the sparse Pochhammer interpolation algorithm and the algorithm of Ben-Or and Tiwari for interpolating polynomials sparse in the standard power basis both illustrate the Dress–Grabmeier framework, the details are quite different. The sparse Pochhammer algorithm differs in a fundamental way from the algorithm of Ben-Or and Tiwari. The counterpart of the auxiliary polynomial $\Psi(z)$ in Ben-Or and Tiwari's algorithm has as roots a^{e_i}, whereas here the auxiliary polynomial $\Psi(z)$ has as roots the *degrees* e_i of the nonzero terms in the interpolating polynomial. This has implications on how we actually compute the roots of the auxiliary polynomial. Also, this algorithm uses a completely different set of evaluation points (importantly, the evaluation points are smaller).

The analysis will be brief, pointing out the effect of the differences just mentioned. We assume unit cost for a black box query. The $f^{(i)}(a)$ for $i = 0, 1, \ldots, 2t-1$ can be computed from $f(a+i)$, $i = 0, 1, \ldots, 2t-1$ by using $O(t^2)$ field operations. The Toeplitz system of equations $\mathcal{F}\vec{\psi} = -\vec{f}$ can be solved using $O(t^2)$ field operations using the Berlekamp–Massey algorithm (Kaltofen and Lakshman (1988)) to find the coefficients of the auxiliary polynomial $\Psi(z)$. We can use the Cantor–Zassenhaus algorithm to factor $\Psi(z)$ modulo a suitable prime p and the Hensel lifting to recover the integer roots. The cost is $O(t^2 + t \log d)$, where d is the degree of the interpolating polynomial (see, for instance, Loos (1983)). The final step of solving a transposed Vandermonde system of equations can be done using the algorithm of Zippel (1990) in $O(t^2)$ field operations. We can now add up the costs of the four steps and we have the following theorem.

THEOREM 2. *The sparse Pochhammer interpolation algorithm performs $O(t^2 + t \log d)$ field operations to interpolate a t-Pochhammer-sparse polynomial given by a black box. Here, d is the degree of the polynomial being interpolated (not part of the input). The algorithm makes $2t$ queries to the black box evaluating the polynomial. The numerator and denominator of the rational numbers that arise during the algorithm have $O(d \log(t))$ bits.* □

2. Sparse interpolation in the Chebyshev basis. In this section, we describe an efficient algorithm for interpolating polynomials in $\mathbf{Q}[x]$ that are sparse in the Chebyshev basis, where $\mathbf{Q}$ is the field of rational numbers. The algorithm is entirely rational, so it applies over any field of characteristic zero. The analysis in §3 in terms of arithmetic operations would apply in the general characteristic zero case as well, provided the evaluation point a is rational and $a \geq 2$. The bit complexity analysis, of course, is specific to the base field $\mathbf{Q}$. Let $T_n(x)$ denote the nth Chebyshev polynomial of the first kind:

$$T_0(x) = 1, \quad T_1(x) = x, \quad T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x) \text{ for } n \geq 2.$$

It is well known that $T_i(x)$, $i = 0, 1, 2, \ldots$ is a $\mathbf{Q}$-basis for the polynomial ring $\mathbf{Q}[x]$.

Let $f(x) = \sum_{i=0}^{m} f_i T_i(x)$ be a polynomial in the ring $\mathbf{Q}[x]$. As usual, we say that $f(x)$ is t-sparse (in the Chebyshev basis) if and only if at most t of the f_i's are nonzero.

Suppose we are given a black box that returns the value of $f(a)$ for any $a \in \mathbf{Q}$. We present an algorithm that interpolates $f(x)$ from its values at specially chosen $2t$ points. Our algorithm may be regarded as a generalization of the Ben-Or and Tiwari algorithm (Ben-Or and Tiwari (1988)). We identify two crucial properties that are required for the generalization to work. In the context of Chebyshev polynomials, they are the following items:

- For $m, n \geq 0$, $T_n(T_m(x)) = T_{mn}(x) = T_m(T_n(x))$, i.e., Chebyshev polynomials commute with respect to composition.
- For $m, n \geq 0$, $T_m(x)T_n(x) = 1/2(T_{m+n}(x) + T_{|m-n|}(x))$, i.e., the product of two Chebyshev polynomials is a *fixed* linear form in two others.

These properties are easy consequences of well-known properties of the Chebyshev polynomials. Before describing our interpolation algorithm, we need a statement of uniqueness of a sparse interpolating polynomial.

LEMMA 3. *Let $f(x)$ be a t-sparse polynomial (in the Chebyshev basis). If for some $a > 1$, $f(T_i(a)) = 0$ for $i = 0, 1, 2, \ldots, t-1$, then $f(x)$ is identically zero.*

Proof. Since $f(x)$ is t-sparse, there exist $f_i \in \mathbf{Q}$ and $\delta_i \in \mathcal{Z}_+$, for $i = 1, \ldots, t$ such that

$$f(x) = f_1 T_{\delta_1}(x) + f_2 T_{\delta_2}(x) + \cdots + f_t T_{\delta_t}(x).$$

Now, we can rewrite the fact that $f(T_i(a)) = 0$ for $i = 0, 1, 2, \ldots, t-1$ as $\mathcal{V}\vec{\mathrm{f}} = \vec{\mathbf{0}}$, where

$$\mathcal{V} = \begin{pmatrix} T_{\delta_1}(T_0(a)) & T_{\delta_2}(T_0(a)) & \ldots & T_{\delta_t}(T_0(a)) \\ T_{\delta_1}(T_1(a)) & T_{\delta_2}(T_1(a)) & \ldots & T_{\delta_t}(T_1(a)) \\ \vdots & \vdots & \ddots & \vdots \\ T_{\delta_1}(T_{t-1}(a)) & T_{\delta_2}(T_{t-1}(a)) & \ldots & T_{\delta_t}(T_{t-1}(a)) \end{pmatrix}, \qquad \vec{\mathrm{f}} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_t \end{pmatrix}$$

If f is not identically zero, then $\mathcal{V}$ is singular. Let $\vec{c} = (c_0, c_1, \ldots, c_{t-1})^T$ be such that $\vec{c}^T\mathcal{V} = 0$. Consider the polynomial

$$C(x) = c_0 T_0(x) + c_1 T_1(x) + \cdots + c_{t-1} T_{t-1}(x)$$

of degree at most $t-1$. Use the fact that $T_{\delta_j}(T_i(a)) = T_i(T_{\delta_j}(a))$ and rewrite $\mathcal{V}$ as

$$\mathcal{V} = \begin{pmatrix} T_0(T_{\delta_1}(a)) & T_0(T_{\delta_2}(a)) & \ldots & T_0(T_{\delta_t}(a)) \\ T_1(T_{\delta_1}(a)) & T_1(T_{\delta_2}(a)) & \ldots & T_1(T_{\delta_t}(a)) \\ \vdots & \vdots & \ddots & \vdots \\ T_{t-1}(T_{\delta_1}(a)) & T_{t-1}(T_{\delta_2}(a)) & \ldots & T_{t-1}(T_{\delta_t}(a)) \end{pmatrix}.$$

Now, $\vec{c}^T\mathcal{V} = 0$ implies that $T_{\delta_1}(a), T_{\delta_2}(a), \ldots, T_{\delta_t}(a)$ are all roots of $C(x)$. These t values are distinct. (In fact, the values $T_i(a)$ are strictly monotonic increasing if $a > 1$, which is straightforward to check from the defining recurrence.) But $C(x)$ is of degree $t-1$ and cannot have t roots. Hence, $\mathcal{V}$ cannot be singular, and f must be identically zero. □

As a consequence, we have the next corollary.

COROLLARY 4. *If $f(x)$ and $g(x)$ are two distinct t-sparse polynomials, then for each $a > 1$ there is an i, $0 \le i \le 2t-1$ such that $f(T_i(a)) \neq g(T_i(a))$.*

Proof. Consider $h(x) = f(x) - g(x)$. The polynomial $h(x)$ is $2t$-sparse and if the corollary is not true, then $h(T_i(a)) = 0$ for $i = 0, \ldots, 2t-1$. Apply the previous lemma. □

We can now describe the sparse interpolation algorithm. As before, $f(x) = \sum_{i=1}^{t} f_i T_{\delta_i}(x)$. The algorithm determines the δ_i first. Once the δ_i are known, it is a simple task to compute the coefficients f_i. Let $a_i = f(T_i(a))$, $i = 0, 1, \ldots, 2t-1$ for some $a > 1$. Consider the polynomial of degree t whose roots are $T_{\delta_i}(a)$ represented in the Chebyshev basis, normalized to have a leading coefficient of 1, i.e.,

$$\Phi(z) = T_t(z) + \phi_{t-1} T_{t-1}(z) + \cdots + \phi_0 T_0(z)$$

and

$$\Phi(T_{\delta_i}(a)) = 0 \text{ for } i = 1, \ldots, t.$$

Our strategy is to compute $\Phi(z)$ and then find its integer roots. First, we show that the coefficients of $\Phi(z)$ satisfy the following linear relations:

LEMMA 5.

$$\sum_{j=0}^{t-1} \phi_j \times (a_{i+j} + a_{|j-i|}) = -(a_{i+t} + a_{|t-i|})$$

for $i = 0, 1, 2, \ldots$.

Proof. Assume $\phi_t = 1$.

$$\begin{aligned}
\left(\sum_{j=0}^{t-1} \phi_j a_{j+i}\right) + a_{t+i} &= \sum_{j=0}^{t} \phi_j \left(\sum_{l=1}^{t} f_l T_{\delta_l}(T_{j+i}(a))\right) \\
&= \sum_{l=1}^{t} f_l \left(\sum_{j=0}^{t} \phi_j T_{\delta_l}(T_{j+i}(a))\right) = \sum_{l=1}^{t} f_l \left(\sum_{j=0}^{t} \phi_j T_{j+i}(T_{\delta_l}(a))\right) \\
&= \sum_{l=1}^{t} f_l \left(\sum_{j=0}^{t} \phi_j (2T_j(T_{\delta_l}(a))T_i(T_{\delta_l}(a)) - T_{|j-i|}(T_{\delta_l}(a)))\right) \\
&= \sum_{l=1}^{t} f_l \left[\Phi(T_{\delta_l}(a)) \times 2T_i(T_{\delta_l}(a)) - \left(\sum_{j=0}^{t} \phi_j T_{|j-i|}(T_{\delta_l}(a))\right)\right] \\
&= -\sum_{l=1}^{t} f_l \left(\sum_{j=0}^{t} \phi_j T_{\delta_l}(T_{|j-i|}(a))\right) \\
&= -\phi_j \left(\sum_{l=1}^{t} f_l T_{\delta_l}(T_{|j-i|}(a))\right) = -\sum_{j=0}^{t} \phi_j a_{|j-i|} \\
&= -\left(\sum_{j=0}^{t-1} \phi_j a_{|j-i|}\right) - a_{|t-i|}.
\end{aligned}$$

Hence, $\sum_{j=0}^{t-1} \phi_j (a_{i+j} + a_{|j-i|}) = -(a_{i+t} + a_{|t-i|})$. □

The above lemma says that the coefficients of $\Phi(z)$ satisfy the system of equations

$$\mathcal{A}\vec{\phi} = -\vec{\alpha},$$

where

$$\mathcal{A} = \begin{pmatrix} 2a_0 & 2a_1 & \dots & 2a_{t-1} \\ 2a_1 & a_2 + a_0 & \dots & a_t + a_{t-2} \\ \vdots & \vdots & \ddots & \vdots \\ 2a_{t-1} & a_t + a_{t-2} & \dots & a_{2t-2} + a_0 \end{pmatrix}, \quad \vec{\phi} = \begin{pmatrix} \phi_0 \\ \phi_1 \\ \vdots \\ \phi_{t-1} \end{pmatrix},$$

$$\vec{\alpha} = \begin{pmatrix} 2a_t \\ a_{t+1} + a_{t-1} \\ \vdots \\ a_{2t-1} + a_1 \end{pmatrix}.$$

Notice that $\mathcal{A}_{i,j} = a_{i+j} + a_{|i-j|}$ is the sum of a Toeplitz matrix and a symmetric Hankel matrix. Next, we show that $\mathcal{A}$ is nonsingular.

LEMMA 6. *$\mathcal{A}$ is nonsingular.*

Proof. Consider the (transposed) *Vandermonde-like* matrix $\mathcal{V}$ and diagonal matrix $\mathcal{B}$.

$$\mathcal{V} = \begin{pmatrix} T_{\delta_1}(T_0(a)) & T_{\delta_2}(T_0(a)) & \dots & T_{\delta_t}(T_0(a)) \\ T_{\delta_1}(T_1(a)) & T_{\delta_2}(T_1(a)) & \dots & T_{\delta_t}(T_1(a)) \\ \vdots & \vdots & \ddots & \vdots \\ T_{\delta_1}(T_{t-1}(a)) & T_{\delta_2}(T_{t-1}(a)) & \dots & T_{\delta_t}(T_{t-1}(a)) \end{pmatrix},$$

$$\mathcal{B} = \begin{pmatrix} 2f_1 & 0 & \dots & 0 \\ 0 & 2f_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 2f_t \end{pmatrix}.$$

According to arguments similar to those in Lemma 1, $\mathcal{V}$ is nonsingular. Since f has exactly t terms, no f_i is zero, hence $\mathcal{B}$ is nonsingular. We claim that $\mathcal{A} = \mathcal{V}\mathcal{B}\mathcal{V}^T$, thereby proving that $\mathcal{A}$ is nonsingular. The (i, j)th element of the product $\mathcal{V}\mathcal{B}\mathcal{V}^{tr}$ is

$$\sum_{l=1}^{t} 2f_l T_{\delta_l}(T_i(a))T_{\delta_l}(T_j(a)) = \sum_{l=1}^{t} 2f_l T_i(T_{\delta_l}(a))T_j(T_{\delta_l}(a))$$

$$= \sum_{l=1}^{t} f_l(T_{i+j}(T_{\delta_l}(a)) + T_{|i-j|}(T_{\delta_l}(a))) = \sum_{l=1}^{t} f_l(T_{\delta_l}(T_{i+j}(a)) + T_{\delta_l}(T_{|i-j|}(a)))$$

$$= a_{i+j} + a_{|i-j|} = \mathcal{A}_{i,j}. \qquad \square$$

From the above lemmas we see that we can compute the coefficients of $\Phi(z)$ by solving the linear system of equations given by $\mathcal{A}\vec{\phi} = -\vec{\alpha}$. The roots of $\Phi(z)$ are integers with special properties and they can be determined without using numerical root finding algorithms. The roots of $\Phi(z)$ are $T_{\delta_i(a)}$ and we can determine the "Chebyshev exponents" δ_i from $T_{\delta_i}(a)$. The coefficients f_i of f can be determined by solving the linear system of equations given by

$$\mathcal{V}\vec{\mathsf{f}} = \vec{\mathsf{a}},$$

where $\mathcal{V}$ is the matrix in the previous lemma and

$$\vec{\mathsf{f}} = (\ f_1 \quad f_2 \quad \ldots \quad f_t\ )^T, \qquad \vec{\mathsf{a}} = (\ a_0 \quad a_1 \quad \ldots \quad a_{t-1}\ )^T.$$

To summarize, we now collect all the steps of our interpolation algorithm.

Algorithm Sparse-Chebyshev-Interpolation

Input: A black box for evaluating a polynomial $f(x)$ known to be t-sparse in the Chebyshev basis.

Output: A list of pairs $(f_i, \delta_i), i = 1, \ldots, t$ such that $f(x) = \sum_{i=1}^{t} f_i T_{\delta_i}(x)$.

1. *Query the black box to obtain the values of* $f(x)$ *at the* $2t$ *points* $x = T_i(a)$, $i = 0, 1, \ldots, 2t-1$ *for some* $a > 1$. *Let* $a_i = f(T_i(a))$.
2. *Solve the system of linear equations* $\mathcal{A}\vec{\phi} = -\vec{\alpha}$. *If*

$$\vec{\phi} = (\ \phi_0 \quad \phi_1 \quad \ldots \quad \phi_{t-1}\ )^T,$$

then the auxiliary polynomial $\Phi(z)$ *is given by*

$$\Phi(z) = z^t + \phi_{t-1}z^{t-1} + \cdots + \phi_1 z + \phi_0.$$

3. *Find the integer roots of* $\Phi(z)$. *The roots are* $T_{\delta_1}(a), T_{\delta_2}(a), \ldots, T_{\delta_t}(a)$. *Find the "Chebyshev powers"* $\delta_1, \delta_2, \ldots, \delta_t$ *from* $T_{\delta_1}(a), T_{\delta_2}(a), \ldots, T_{\delta_t}(a)$.
4. *Solve the linear system of equations* $\mathcal{V}\vec{\mathsf{f}} = \vec{\mathsf{a}}$ *to obtain the coefficients* $f_1, \ldots, f_t$ *of* f. *Output the list of pairs* $[(f_1, \delta_1), \ldots, (f_t, \delta_t)]$.

3. Analysis of the Chebyshev Interpolation Algorithm. In our analysis, we first count the number of field operations ($+, -, \times, /$ involving rational numbers) performed by the algorithm in the worst case. We then bound the bit sizes of the rational numbers that might arise during the computation, thus giving upper bounds on the number of bit operations performed by the algorithm. We also assume that querying the black box for the value of $f(x)$ is a unit cost operation.

Solving $\mathcal{A}\vec{\phi} = -\vec{\alpha}$. The matrix $\mathcal{A}$ is the sum of a Hankel and a Toeplitz matrix of size $t \times t$. Such systems can be solved by well-known techniques using $O(t^2)$ field operations (see, for example, Merchant and Parks (1982)).

Finding the integer roots of $\Phi(z)$. The special nature of the roots of $\Phi(z)$ (all are integers of the type $T_{\delta_j}(a)$) allows us to compute them without using regular root finders. Since $\Phi(z)$ is expressed in the Chebyshev basis with leading coefficient 1, ϕ_{t-1}, the coefficient of $T_{t-1}(z)$, is given by

$$\phi_{t-1} = \begin{cases} -T_{\delta_1}(a) & \text{if } t = 1, \\ -2\sum_{j=1}^{t} T_{\delta_j}(a) & \text{if } t > 1. \end{cases}$$

Let $s = -\phi_{t-1}/2$. It is easily shown from the defining recurrence that $T_n(a) = 1/2(\beta^n + \bar{\beta}^n)$, where $\beta = a + \sqrt{a^2 - 1}$ and $\bar{\beta} = a - \sqrt{a^2 - 1}$ for $n = 0, 1, 2, \ldots$. Suppose that for convenience we choose $a = 2$. Then $\beta = 2 + \sqrt{3}$ and $\bar{\beta} = 2 - \sqrt{3}$. Without any loss of generality, assume that $\delta_1 > \delta_2 > \cdots > \delta_t$. Clearly, δ_1 is the unique integer k such that

$$1/2(\beta^k + \bar{\beta}^k) \leq s < 1/2(\beta^{k+1} + \bar{\beta}^{k+1}).$$

We can find such a k by an algorithm analogous to the binary powering algorithm as follows:

Choose $D = 2\lceil \log_2(s) \rceil$; $(1/2(\beta^D + \bar{\beta}^D) > s)$
Compute integer arrays p, q such that
  $p_i + q_i\sqrt{3} = \beta^{2^i}$ for $i = 0, 1, \ldots$ to at most D.
$i := 0$; $p_0 := 1$; $q_0 := 0$;
loop
  $increment(i)$;
  $p_i := p_{i-1}^2 + 3q_{i-1}^2$; $q_i := 2p_{i-1}q_{i-1}$;
until $p_i > s$;
$decrement(i)$;
$k := 2^i$; $j := i - 1$; $\widehat{p} := p_i$; $\widehat{q} := q_i$;
loop
  $tmp := \widehat{p} \times p_i + 3\widehat{q} \times q_i$;
  if $(tmp \leq s)$ **then**
    $\widehat{q} := p_i \times \widehat{q} + q_i \times \widehat{p}$; $\widehat{p} := tmp$; $k := k + 2^j$; **fi;**
  $decrement(j)$;
until $j = 0$;
return $k, \widehat{p}$; $\{\widehat{p} = T_k(2)\}$

Once we know k, i.e., δ_1, we can set $s := s - T_k(2)$ and repeat the process to get δ_2 and so on. The number of field operations performed by the above algorithm is $O(\log_2 D)$ and we need to run it t times to compute all the δ_i. Therefore, the Chebyshev exponents can be computed by performing $O(t \log_2 D)$ field operations. D is chosen to be $2\lceil \log_2(s) \rceil$ and

$$s = -\phi_{t-1}/2 = \sum_{j=1}^{t} T_{\delta_j}(a) < 1/2(\beta^{\delta_1+1} + \bar{\beta}^{\delta_1+1}) < 4^{\delta_1}.$$

Therefore, the number of bits in the integers involved in the root finding step is no more than $2\delta_1$.

Solving $\mathcal{V}\vec{\mathsf{f}} = \vec{\mathsf{a}}$. $\mathcal{V}$ is the transpose of a Vandermonde-like matrix. An $n \times n$ non-singular Vandermonde matrix can be inverted by using $O(n^2)$ field operations and $O(n)$ space (Zippel (1990)). More efficient algorithms ($O(n \log^2 n \log(\log n))$ time) that use fast polynomial multiplication are also known (Canny, Kaltofen, and Lakshman (1989)). In the following discussion, we adapt Zippel's algorithm for solving the system of equations $\mathcal{V}\vec{\mathsf{f}} = \vec{\mathsf{a}}$. Consider the product

$$\mathcal{V}^T\mathcal{A} = \begin{pmatrix} T_0(T_{\delta_1}(a)) & T_1(T_{\delta_1}(a)) & \dots & T_{t-1}(T_{\delta_1}(a)) \\ T_0(T_{\delta_2}(a)) & T_1(T_{\delta_2}(a)) & \dots & T_{t-1}(T_{\delta_2}(a)) \\ \vdots & \vdots & \ddots & \vdots \\ T_0(T_{\delta_t}(a)) & T_1(T_{\delta_t}(a)) & \dots & T_{t-1}(T_{\delta_t}(a)) \end{pmatrix} \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1t} \\ c_{21} & c_{22} & \dots & c_{2t} \\ \vdots & \vdots & \ddots & \vdots \\ c_{t1} & c_{t2} & \dots & c_{tt} \end{pmatrix}.$$

This product is the matrix

$$\begin{pmatrix} P_1(T_{\delta_1}) & P_2(T_{\delta_1}) & \dots & P_t(T_{\delta_1}) \\ P_1(T_{\delta_2}) & P_2(T_{\delta_2}) & \dots & P_t(T_{\delta_2}) \\ \vdots & \vdots & \ddots & \vdots \\ P_1(T_{\delta_t}) & P_2(T_{\delta_t}) & \dots & P_t(T_{\delta_t}) \end{pmatrix},$$

where $P_j(z)$ is the polynomial

$$P_j(z) = c_{1j}T_0(z) + c_{2j}T_1(z) + c_{3j}T_2(z) + \cdots + c_{tj}T_{t-1}(z).$$

Notice that by choosing

$$P_j(z) = \prod_{i \neq j} \frac{z - T_{\delta_i}(a)}{T_{\delta_j}(a) - T_{\delta_i}(a)},$$

the product becomes the identity matrix. In other words, the jth column of the inverse of $\mathcal{V}^T$ is made up of the coefficients of $P_j(z)$ *expressed in the Chebyshev basis.* Let $P_{j,i}$ denote the coefficient of $T_i(z)$ in $P_j(z)$. Since the inverse of the transpose is the transpose of the inverse, the system of equations $\mathcal{V}\vec{\mathsf{f}} = \vec{\mathsf{a}}$, i.e.,

$$\begin{pmatrix} T_{\delta_1}(T_0(a)) & T_{\delta_2}(T_0(a)) & \dots & T_{\delta_t}(T_0(a)) \\ T_{\delta_1}(T_1(a)) & T_{\delta_2}(T_1(a)) & \dots & T_{\delta_t}(T_1(a)) \\ \vdots & \vdots & \ddots & \vdots \\ T_{\delta_1}(T_{t-1}(a)) & T_{\delta_2}(T_{t-1}(a)) & \dots & T_{\delta_t}(T_{t-1}(a)) \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_t \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{t-1} \end{pmatrix},$$

has the solution

$$f_j = a_0P_{j,0} + a_1P_{j,1} + \cdots + a_{t-1}P_{j,t-1} \quad \text{for} \quad j = 1, \dots, t.$$

The coefficients $P_{j,i}$ can be computed as follows. Let

$$P(z) = \prod_{k=1}^{t}(z - T_{\delta_k}(a)).$$

($P(z)$ is actually $\Phi(z)/2^{t-1}$, $\Phi(z)$ being the auxiliary polynomial computed in step 2 of the interpolation algorithm.) Let

$$\widehat{P}_j(z) = P(z)/(z - T_{\delta_j}(a)),$$

be expressed in the Chebyshev basis. We can compute $\widehat{P}_j(z)$ from $P(z)$ by "synthetic Chebyshev division"—use the identity

$$T_k(z) = (z - T_{\delta_j}(a)) \times 2T_{k-1}(z) + 2T_{\delta_j}(a)T_{k-1}(z) + T_{k-2}(z) \quad \text{for } k > 1$$

$t - 1$ times on $P(z)$. This division can be performed by using $O(t)$ field operations. Let $D_j = \widehat{P}_j(T_{\delta_j}(a))$. Clearly,

$$P_j(z) = \widehat{P}_j(z)/D_j.$$

D_j can be computed by using $O(t)$ field operations (evaluate $\widehat{P}_j(z)$ at $T_{\delta_j}(a)$). Therefore, we can compute all the Chebyshev coefficients of $P_j(z)$ in $O(t)$ field operations, and from them we can compute f_j by using $O(t)$ field operations. Since each f_j needs $O(t)$ field operations in this scheme, we need $O(t^2)$ field operations to compute all the f_j, $j = 1, \ldots, t$. Notice that we need $O(t)$ space to hold the coefficients of each $P_j(z)$. However, we can reuse the same space for all the $P_j(z)$. Hence, we only need $O(t)$ storage units. Also, notice that the rational numbers encountered in this scheme have numerators and denominators with $O(t \log(T_{\delta_1}(a)))$ bits. Notice that $\log(T_{\delta_1}(a) = O(d))$, where d is the degree of the interpolating polynomial. We can now add up the costs of the four steps and we have the next theorem.

THEOREM 7. *The sparse Chebyshev interpolation algorithm performs $O(t^2 + t \log d)$ field operations to interpolate a t-Chebyshev-sparse polynomial given by a black box. Here d is the degree of the polynomial being interpolated (not part of the input). The algorithm makes $2t$ queries to the black box evaluating the polynomial. The numerator and denominator of the rational numbers that arise during the algorithm have $O(td)$ bits.* □

Remark 1. We may not know the exact number of terms in the polynomial, just only an upper bound $\tau \geq t$. In such a case, we can still use the algorithm, except that the matrix $\mathcal{A}$ will be singular (for $\tau > t$). The solution is to use the largest nonsingular leading principle minor of $\mathcal{A}$ in place of $\mathcal{A}$. Its rank gives the exact number of terms in the interpolating polynomial. The details are worked out for the Ben-Or and Tiwari algorithm in Kaltofen and Lakshman (1988) and are essentially the same for this case.

Remark 2. We have used classical algorithms for solving the various subproblems in the interpolation algorithm. One can use asymptotically faster algorithms to improve the overall complexity of the interpolation algorithm to $O(t(\log^2 t \log(\log t) + \log d))$ field operations. See Kaltofen and Lakshman (1988) and Canny et al. (1989).

Remark 3. The algorithm that we have discussed can be viewed as a generalization of the Ben-Or and Tiwari algorithm which in turn is based on the BCH decoding algorithm. The generalization uses a crucial property of the Chebyshev polynomials, namely, that they commute with respect to composition, i.e., $T_n(T_m(z)) = T_m(T_n(z))$. It turns out that this property is rather special and the only *families* of polynomials that exhibit this property are the Chebyshev polynomials and the standard powers $\{1, z, z^2, \ldots\}$. More precisely, it is known that if in a family of polynomials $f_0(z), f_1(z), f_2(z), \ldots$ with $\deg(f_i(z)) = i$, it is true that $f_i(f_j(z)) = f_j(f_i(z))$ for any $i, j \geq 0$, then either $f_i(z) = \lambda((\lambda^{-1}(z))^i)$ for all i or $f_i(z) = \lambda(T_i(\lambda^{-1}(z)))$ for all i, where $\lambda(z) = az + b$ and $\lambda^{-1}(z) = (z - a)/b$ for some constants a, b. (See, for instance, Rivlin (1974).)[1] The algorithm can be adapted easily to interpolate polynomials that are sparse in such "generalized" power bases or Chebyshev bases.

One can consider several possible "natural" bases and polynomials that are sparse in those bases. For example, suppose we know that the polynomial given by a black box is t-sparse in the basis $1, x - \alpha, (x - \alpha)^2, (x - \alpha)^3, \ldots$ for some unknown α. Two recent papers discuss algorithms for solving this problem (Grigoriev and Karpinski (1992) and Lakshman and Saunders (1994)).

Sparsity in the standard basis is related to the number of real roots of the polynomial (sparsity limits the number of variations in sign), which is not the case when we consider polynomials that are sparse in the Chebyshev or Pochhammer bases. What are the corresponding observations about sparsity in these bases?

Acknowledgments. We thank our colleagues Prof. Bobby Caviness, Prof. David Wood, and Prof. Jeremy Johnson for several helpful discussions. Our thanks also to Prof. Michael Singer of North Carolina State University for detailed comments on an earlier draft.

[1] We are grateful to Dr. Jeremy Johnson of Drexel University for bringing this to our attention.

REFERENCES

M. Ben Or and P. Tiwari, (1988), *A deterministic algorithm for sparse multivariate polynomial interpolation*, Proc. 20th Symp. Theory of Computing, ACM Press, New York, pp. 301–309.

A. Borodin and P. Tiwari, (1990), *On the decidability of sparse univariate polynomial interpolation*, Proc. 22nd Symp. Theory of Computing, ACM Press, New York, pp. 535–545.

J. F. Canny, E. Kaltofen, and Y. N. Lakshman, (1989), *Solving systems of non-linear equations faster*, Proc. ISSAC 1989, Portland, OR, ACM Press, New York, pp. 121–128.

M. Clausen, A. Dress, J. Grabmeier, and M. Karpinski, (1988), *On zero testing and interpolation of k-sparse multivariate polynomials over finite fields*, Tech. Rep. 88.06.006, IBM Germany, Heidelberg Scientific Center, June 1988.

A. Dress and J. Grabmeier, (1991), *The interpolation problem for k-sparse polynomials and Character sums*, Adv. in Appl. Math, 12, pp. 57–75.

D. Grigoriev and M. Karpinski, (1987), *The matching problem for bipartite graphs with polynomially bounded permanents is in* NC, Proc. 28th IEEE Symp. Foundations Comp. Sci., pp. 166–172.

D. Grigoriev, M. Karpinski, and M. Singer, (1990), *Fast parallel algorithms for sparse multivariate polynomial interpolation over finite fields*, SIAM J. Comput., 19, pp. 1059–1063.

———, (1991a), *The interpolation problem for k-sparse sums of eigenfunctions of operators*, Adv. in Appl. Math., 12, pp. 76–81.

———, (1994), *Computational complexity of sparse rational interpolation*, SIAM J. Comput., 23, pp. 1–11.

D. Grigoriev and M. Karpinski, (1992), *A zero-test and an interpolation algorithm for the shifted sparse polynomials*, Proc. AAECC 1993, Puerto-Rico, Lecture Notes in Comput. Sci. 673, Springer-Verlag, New York, pp. 162–169.

D. Grigoriev, M. Karpinski, and A. M. Odlyzko, (1992), *Existence of short proofs of non-divisibility of sparse polynomials under the extended Riemann hypothesis*, Proc. International Symposium on Symbolic and Algebraic Computation 92, ACM Press, New York, pp. 117–122.

E. Kaltofen and Y. N. Lakshman, (1988), *Improved sparse multivariate polynomial interpolation algorithms*, Proc. International Symposium on Symbolic and Algebraic Computation 1988, Rome, Italy, Lecture Notes in Comput. Sci. 358, Springer-Verlag, New York, pp. 467–474.

E. Kaltofen and B. Trager, (1990), *Computing with polynomials given by black boxes for their evaluations: Greatest common divisors, factorization, separation of numerators and denominators*, J. Symbolic Comput., 9, pp. 301–320.

Y. N. Lakshman and B. D. Saunders, (1994), *On Computing sparse shifts for univariate polynomials*, Proc. International Symposium on Symbolic and Algebraic Computation 94, ACM Press, New York, pp. 108–113.

R. Loos, (1983), *Computing rational zeros of integral polynomials by p-adic expansion*, SIAM J. Comput., 12, pp. 286–293.

G. A. Merchant and T. M. Parks, (1982), *Efficient solution of a Toeplitz-plus-Hankel matrix system of equations*, IEEE Trans. Acoustics, Speech, and Signal Proc., 30, pp. 40–44.

T. Rivlin, (1974), *The Chebyshev polynomials*, John Wiley, New York.

R. Zippel, (1990), *Interpolating polynomials from their values*, J. Symbolic Comput., 9, pp. 375–403.

SIAM J. COMPUT.
Vol. 24, No. 2, pp. 398–410, April 1995

015

A NEW APPROACH TO FORMAL LANGUAGE THEORY BY KOLMOGOROV COMPLEXITY*

MING LI† AND PAUL VITÁNYI‡

Abstract. We present a new approach to formal language theory by using Kolmogorov complexity. The main results presented here are an alternative for pumping lemma(s), a new characterization for regular languages, and a new method to separate deterministic context-free languages and nondeterministic context-free languages. The use of the "incompressibility arguments" is illustrated by many examples. The approach is also successful at the high end of the Chomsky hierarchy since one can quantify nonrecursiveness in terms of Kolmogorov complexity.

Key words. formal language theory, Kolmogorov complexity, pumping lemmas, regular languages, finite automata, deterministic context-free languages

AMS subject classifications. 68Q30, 68Q45, 68Q68, 68Q50

1. Introduction.

It is feasible to reconstruct parts of formal language theory by using algorithmic information theory (Kolmogorov complexity). We provide theorems on how to use Kolmogorov complexity as a concrete and powerful tool. We do not just want to introduce fancy mathematics; our goal is to help our readers do a large part of formal language theory in the most essential, easiest, and sometimes even obvious ways. In this paper, it is only important to us to demonstrate that the application of Kolmogorov complexity in the targeted area is not restricted to trivialities. The proofs of the theorems in this paper may not be easy. However, the theorems are the type that are used as a tool. Once derived, our theorems are easy to apply.

1.1. Prelude.

The first application of Kolmogorov complexity in the theory of computation was in [18] and [19]. By redoing proofs of known results, it was shown that static, descriptional (program size) complexity of a *single* random string can be used to obtain lower bounds on dynamic, computational (running time) complexity. None of the inventors of Kolmogorov complexity originally had these applications in mind. Recently, Kolmogorov complexity has been applied extensively to solve classic open problems of sometimes two decades standing [15], [11], [8], [9]. For more examples, see the textbook [12].

The secret of Kolmogorov complexity's success in dynamic, computational lower bound proofs rests on a simple fact: the overwhelming majority of strings has hardly any computable regularities. We call such a string "Kolmogorov random" or "incompressible." A Kolmogorov random string cannot be (effectively) compressed. Incompressibility is a noneffective property: no individual string, except finitely many, can be proved incompressible.

Recall that a traditional lower bound proof by counting usually involves *all* inputs of certain length. One shows that a certain lower bound has to hold for *some* "*typical*" input. Since an individual typical input is *hard* (sometimes impossible) to find, the proof must involve all the inputs. Now we understand that a typical input of each length can be constructed via

*Received by the editors February 22, 1993; accepted for publication November 12, 1993. A preliminary version of part of this work was presented at the 16th *International Colloquium on Automata, Languages, and Programming*, Stresa, Italy, July 1989.

†Current address, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1 (mli@math.waterloo.edu). The research of this author was supported in part by National Science Foundation grant DCR-8606366, Office of Naval Research grant N00014-85-k-0445, Army Research Office grant DAAL03-86-K-0171, and by NSERC operating grants OGP-0036747 and OGP-046506. Part of this work was performed while the author was at the Department of Computer Science, York University, North York, Ontario, Canada.

‡CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands (paulv@cwi.nl). Partially supported by NSERC International Scientific Exchange Award ISE0046203, and by NWO through NFI Project ALADDIN under contract NF 62-376.

an incompressible string. However, only finitely many individual strings can be effectively proved to be incompressible. No wonder the old counting arguments had to involve all inputs. In a proof using the "incompressibility method," one uses an individual incompressible string that is known to *exist* even though it cannot be constructed. Then one shows that if the assumed lower time bound would not hold, then this string could be compressed, and hence it would not be incompressible.

1.2. Outline of the paper. The incompressibility argument also works for formal languages and automata theory. Assume the basic notions treated in a textbook such as [6].

The first result is a powerful alternative to pumping lemmas for regular languages. It is well known that not all nonregular languages can be shown to be nonregular by the usual uvw-pumping lemma. There is a plethora of pumping lemmas to show nonregularity, like the "marked pumping lemma," and so on. In fact, it seems that many example nonregular languages require their own special purpose pumping lemmas. Recently, [7], [21], [3], exhaustive pumping lemmas that characterize the regular languages have been obtained.

These pumping lemmas are complicated and complicated to use. The last reference uses Ramsey theory. In contrast, by using Kolmogorov complexity we give a new characterization of the regular languages that simply makes our intuition of the "finite stateness" of these languages rigorous and easy to apply. Since it is a characterization, it works for all nonregular languages. We give several examples of its application, some of which were quite difficult using pumping lemmas.

To prove that a certain context-free language (cfl) is not deterministic context-free (dcfl) has required laborious ad-hoc proofs [6], or cumbersome and difficult pumping lemmas or iteration theorems [4], [24]. We give necessary (Kolmogorov complexity) conditions for dcfl, that are very easy to apply. We test the new method on several examples in cfl–dcfl, which were hard to handle before. In certain respects the KC-DCFL lemma may be more powerful than the related lemmas and theorems mentioned above. On the high end of the Chomsky hierarchy we present, for completeness, a known characterization of recursive languages, and a necessary condition for recursively enumerable languages.

2. Kolmogorov complexity. From now on, let x denote both the natural number and the xth binary string in the sequence 0, 1, 00, 01, 10, 11, 000, ... That is, the representation "3" corresponds both to the natural number 3 and to the binary string 00. This way we obtain a natural bijection between the nonnegative integers $\mathcal{N}$ and the finite binary strings $\{0, 1\}^*$. Numerically, the binary string $x_{n-1} \ldots x_0$ corresponds to the integer

$$2^n - 1 + \sum_{i=0}^{n-1} x_i 2^i. \tag{1}$$

We use notation $l(x)$ to denote the *length* (number of bits) of a binary string x. If x is not a finite binary string but another finite object like a finite automaton, a recursive function, or a natural number, then we use $l(x)$ to denote the length of its standard binary description. Let $\langle \cdot, \cdot \rangle : \mathcal{N} \times \mathcal{N} \to \mathcal{N}$ be a standard recursive, invertible, one-one encoding of pairs of natural numbers in natural numbers. This idea can be iterated to obtain a pairing from triples of natural numbers with natural numbers $\langle x, y, z \rangle = \langle x, \langle y, z \rangle \rangle$, and so on.

Any of the usual definitions of Kolmogorov complexity in [10], [19], and [12] will do for the sequel. We are interested in the shortest effective description of a finite object x. To fix thoughts, consider the problem of describing a string x over 0's and 1's. Let $T_1, T_2, \ldots$ be the standard enumeration of Turing machines. Since T_i computes a partial recursive function $\phi_i : \mathcal{N} \to \mathcal{N}$, we obtain the standard enumeration $\phi_1, \phi_2, \ldots$ of partial recursive functions.

We denote $\phi(\langle x, y\rangle)$ as $\phi(x, y)$. Any partial recursive function ϕ from strings over 0's and 1's to such strings, together with a string p, the *program* for ϕ to compute x, such that $\phi(p) = x$, is a description of x. It is useful to generalize this idea to the conditional version: $\phi(p, y) = x$ such that p is a program for ϕ to compute x, given a binary string y for free. Then the *descriptional* complexity C_ϕ of x, *relative* to ϕ and y, is defined by

$$C_\phi(x|y) = \min\{l(p) : p \in \{0, 1\}^*, \phi(p, y) = x\},$$

or ∞ if no such p exists.

For a *universal* partial recursive function ϕ_0, computed by the universal Turing machine U, we know that, for each partial recursive function $\phi = Q_i$ there is a constant c_ϕ such that for all strings x, y, we have $\phi_0(i, x, y) = \phi(x, y)$. Hence, $C_{\phi_0}(x|y) \leq C_\phi(x|y) + c_\phi$. We fix a reference universal function ϕ_0 and define the *conditional Kolmogorov complexity* of x given y as $C(x|y) = C_{\phi_0}(x|y)$.[1]

The *unconditional Kolmogorov complexity* of x is $C(x) = C(x|\epsilon)$, where ϵ denotes the empty string ($l(\epsilon) = 0$).

Since there is a Turing machine that just copies its input to its output we have $C(x|y) \leq l(x) + O(1)$, for each x and y. Since there are 2^n binary strings of length n, but only $2^n - 1$ possible shorter descriptions, it follows that $C(x) \geq l(x)$ for some binary string x of each length. We call such strings *incompressible* or *random* [16], [17]. It also follows that for each length n and each binary string y, there is a binary string x of length n such that $C(x|y) \geq l(x)$. Considering C as an integer function, using the obvious one-one correspondence between finite binary words and nonnegative integers, it can be shown that $C(x) \to \infty$ for $x \to \infty$. Finally, $C(x, y)$ denotes $C(\langle x, y\rangle)$.

EXAMPLE 1 (self-delimiting strings). A *prefix code* is a mapping from finite binary code words to source words such that no code word is a proper prefix of any other code word. We define a particular prefix code.

For each binary source word $x = x_1 \ldots x_n$, define the code word $\bar{x}$ by

$$\bar{x} = 1^{l(x)}0x.$$

Define

$$x' = \overline{l(x)}x.$$

The string x' is called the *self-delimiting* code of x.

Set $x = 01011$. Then $l(x) = 5$, which corresponds to binary string "10," and $\overline{l(x)} = 11010$. Therefore, $x' = 1101001011$ is the self-delimiting code of "01011."

The self-delimiting code of a positive integer x requires $l(x) + 2\log l(x) + 1$ bits. It is easy to verify that $l(x) = \lfloor \log(x + 1) \rfloor$. All logarithms are base 2 unless otherwise noted. For convenience, we simply replace the length $l(x)$ of a natural number x by "$\log x$."

EXAMPLE 2 (substrings of incompressible strings). Is a substring of an incompressible string also incompressible? A string $x = uvw$ can be specified by a short description for v of length $C(v)$, a description of $l(u)$, and the literal description of uw. Moreover, we need information to tell these three items apart. Such information can be provided by prefixing each item with a self-delimiting description of its length. Together this takes $C(v) + l(uw) + O(\log l(x))$ bits. Hence,

$$C(x) \leq C(v) + O(\log l(x)) + l(uw).$$

[1] Similarly, we define the complexity of the xth partial recursive function ϕ conditional to the yth partial recursive function ψ by $C(\phi|\psi) = C(x|y)$.

Thus, if we choose x incompressible, $C(x) \geq l(x)$, then we obtain

$$C(v) \geq l(v) - O(\log l(x)).$$

It can be shown that this is optimal—some substring of an incompressible string of length n may be compressible by an $\Omega(\log n)$ additional term. This conforms to a fact we know from probability theory: every random string of length n is expected to contain a run of about $\log n$ consecutive zeros (or ones). Such a substring has complexity $O(\log \log n)$.

3. Regular sets and finite automata.

DEFINITION 3.1. *Let Σ be a finite nonempty alphabet, and let Q be a (possibly infinite) nonempty set of states. A* transition function *is a function $\delta : \Sigma \times Q \to Q$. We extend δ to δ' on Σ^* by $\delta'(\epsilon, q) = q$ and*

$$\delta'(a_1 \ldots a_n, q) = \delta(a_n, \delta'(a_1 \ldots a_{n-1}, q)).$$

Clearly, if δ' is not $1-1$, then the automaton "forgets" because some x and y from Σ^* drive δ' into the same memory state. An automaton *A is a quintuple $(\Sigma, Q, \delta, q_0, q_f)$, where everything is as above and $q_0, q_f \in Q$ are distinguished as* initial state *and* final state, *respectively. We call A a* finite automaton *(fa) if Q is finite.*

We denote "indistinguishability" of a pair of histories $x, y \in \Sigma^*$ by $x \sim y$, defined by $\delta'(x, q_0) = \delta'(y, q_0)$. "Indistinguishability" of strings is reflexive, symmetric, transitive, and right-invariant ($\delta'(xz, q_0) = \delta'(yz, q_0)$ for all z). Thus, "indistinguishability" is a right-invariant equivalence relation on Σ^*. It is a simple matter to ascertain this formally.

DEFINITION 3.2. *The language accepted by automaton A as above is the set $L = \{x : \delta'(x, q_0) = q_f\}$. A* regular language *is a language accepted by a finite automaton.*

It is a straightforward exercise to verify from the definitions the following fact (which will be used later).

THEOREM 3.3 (Myhill and Nerode). *The following statements about $L \subseteq \Sigma^*$ are equivalent.*

(i) *$L \subseteq \Sigma^*$ is accepted by some finite automaton.*

(ii) *L is the union of equivalence classes of a right-invariant equivalence relation of finite index on Σ^*.*

(iii) *For all $x, y \in \Sigma^*$ define right-invariant equivalence $x \sim y$ by the following item: for all $z \in \Sigma^*$ we have $xz \in L$ iff $yz \in L$. Then the number of $\sim$-equivalence classes is finite.*

Subsequently, closure of finite automaton languages under complement, union, and intersection follow by simple construction of the appropriate δ functions from given ones. Details can be found in any textbook on the subject such as [6]. The clumsy pumping lemma approach can now be replaced by the Kolmogorov formulation below.

3.1. Kolmogorov complexity replacement for the pumping lemma. An important part of formal language theory is deriving a hierarchy of language families. The main division is the Chomsky hierarchy, with regular languages, context-free languages, context-sensitive languages and recursively enumerable languages. The common way to prove that certain languages are not regular is by using "pumping" lemmas, for instance, the uvw lemma. However, these lemmas are quite difficult to state and cumbersome to prove or use. In contrast, we show below how to replace such arguments by simple and intuitive, yet rigorous, Kolmogorov complexity arguments.

Regular languages coincide with the languages accepted by finite automata. This invites a straightforward application of Kolmogorov complexity. Let us give an example. We prove that $\{0^k 1^k : k \geq 1\}$ is not regular. If it were, then the state q of a particular accepting fa A after processing 0^k, together with A, is, up to a constant, a description of k. Namely, by running A

initialized in state q on input consisting of only 1's, the first time A enters an accepting state is after precisely k consecutive 1's. The size of the description of A and q is bounded by a constant, say c, which is independent of k. Altogether, it follows that $C(k) \leq c + O(1)$. But choosing k with $C(k) \geq \log k$ we obtain a contradiction for all large enough k. Hence, since A has a fixed finite number of states, there is a fixed finite number that bounds the Kolmogorov complexity of each natural number: contradiction. We generalize this observation as follows.

DEFINITION 3.4. *Let Σ be a finite nonempty alphabet and $\phi : \mathcal{N} \to \Sigma^*$ be a total recursive function. Then ϕ enumerates (possibly a proper subset of) Σ^* in order $\phi(1), \phi(2), \ldots$. We call such an order* effective *and ϕ an* enumerator. *The* lexicographical order *is the effective order such that all words in Σ^* are ordered first according to length, and then lexicographically within the group of each length. Another example is ϕ such that $\phi(i) = p_i$, the standard binary representation of the ith prime, is an effective order in $\{0, 1\}^*$. In this case ϕ does not enumerate all of Σ^*. Let $L \subseteq \Sigma^*$. Define $L_x = \{y : xy \in L\}$.*

LEMMA 3.5 (KC regularity). *Let $L \subseteq \Sigma^*$ be regular and let ϕ be an enumerator in Σ^*. Then there exists a constant c that depends only on L and ϕ such that for each x, if y is the nth string enumerated in (or in the complement of) L_x, then $C(y) \leq C(n) + c$.*

Proof. Let L be a regular language. The nth string y such that $xy \in L$ for some x can be described by

- this discussion and a description of the fa that accepts L,
- a description of ϕ, and
- the state of the fa after processing x, and the number n.

The statement "or in the complement of" follows, since regular languages are closed under complementation. □

As an application of the KC Regularity lemma we prove that $\{1^p : p \text{ is prime}\}$ is not regular. Consider the string $xy = 1^p$ with p the $(k+1)$th prime. Set $x = 1^{p'}$, with p' the kth prime. Then $y = 1^{p-p'}$, and y is the lexicographical first element in L_x. Hence, by Lemma 3.5, $C(p - p') = O(1)$. But the difference between two consecutive primes grows unbounded. Since there are only $O(1)$ descriptions of length $O(1)$, we have a contradiction. We give some more examples from the well-known textbook of Hopcroft and Ullman [6].

EXAMPLE 3 (Exercise 3.1(h)* in [6]). Show that $L = \{xx^R w : x, w \in \{0, 1\}^* - \{\epsilon\}\}$ is not regular. Set $x = (01)^m$, where $C(m) \geq \log m$. Then, the lexicographically first word in L_x is y with $y = (10)^m 0$. But $C(y) = \Omega(\log m)$, which contradicts the KC Regularity lemma.

EXAMPLE 4. Prove that $L = \{0^i 1^j : i \neq j\}$ is not regular. Set $x = 0^m$, where $C(m) \geq \log m$. Then the lexicographically first word *not* in $L_x \bigcap \{1\}^*$ is $y = 1^m$. But $C(y) = \Omega(\log m)$, which contradicts the KC Regularity lemma.

EXAMPLE 5 (Exercise 3.6* in [6]). Prove that $L = \{0^i 1^j : \gcd(i, j) = 1\}$ is not regular. Set $x = 0^{(p-1)!}1$, where $p > 3$ is a prime, $l(p) = n$, and $C(p) \geq n - \log n$. Then the lexicographically first word in L_x is 1^{p-1}, which contradicts the KC Regularity lemma.

EXAMPLE 6 (§2.2, Exercises 11–15 in [4]). Prove that $\{p : p$ is the standard binary representation of a prime $\}$ is not regular. Suppose the contrary, and p_i denotes the ith prime, $i \geq 1$. Consider the least binary $p_m = uv$ $(= u2^{l(v)} + v)$, with $u = \Pi_{i<k} p_i$ and v not in $\{0\}^*\{1\}$. Such a prime p_m exists since each interval $[n, n + n^{11/20}]$ of the natural numbers contains a prime [5].

Consider p_m now as an integer, $p_m = 2^{l(v)}\Pi_{i<k} p_i + v$. Since integer $v > 1$ and v is not divided by any prime less than p_k (because p_m is prime), the binary length $l(v) \geq l(p_k)$. Because p_k goes to infinity with k, the value $C(v) \geq C(l(v))$ also goes to infinity with k. But since v is the lexicographical first suffix, with integer $v > 1$ such that $uv \in L$, we have $C(v) = O(1)$ by the KC Regularity lemma, which is a contradiction.

3.2. Kolmogorov complexity characterization of regular languages. While the pumping lemmas are not precise enough (except for the difficult construction in [3]) to characterize the regular languages, this is easy with Kolmogorov complexity. In fact, the KC Regularity lemma is a direct corollary of the characterization below. The theorem is not only a device to show that some nonregular languages are nonregular, as are the common pumping lemmas, but it is a *characterization* of the regular sets. Consequently, it determines whether or not a given language is regular, just like the Myhill–Nerode theorem. The usual characterizations of regular languages seem to be practically useful only to prove regularity. The need for pumping lemmas stems from the fact that characterizations tend to be very hard to use in showing nonregularity. In contrast, the KC characterization is practicable for both purposes, as shown in the examples.

DEFINITION 3.6. *Let Σ be a nonempty finite alphabet, and let y_i be the ith element of Σ^* in lexicographic order, $i \geq 1$. For $L \subseteq \Sigma^*$ and $x \in \Sigma^*$, let $\chi = \chi_1\chi_2 \ldots$ be the* characteristic sequence *of $L_x = \{y : xy \in L\}$, defined by $\chi_i = 1$ if $xy_i \in L$,* and *$\chi_i = 0$* otherwise. We denote $\chi_1 \ldots \chi_n$ by $\chi_{1:n}$.

THEOREM 3.7 (Regular KC characterization). *Let $L \subseteq \Sigma^*$, and assume the notation above. The following statements are equivalent.*

(i) *L is regular.*

(ii) *There is a constant c_L that depends only on L, such that for all $x \in \Sigma^*$ and for all n, $C(\chi_{1:n}|n) \leq c_L$.*

(iii) *There is a constant c_L that depends only on L, such that for all $x \in \Sigma^*$ and for all n, $C(\chi_{1:n}) \leq C(n) + c_L$.*

(iv) *There is a constant c_L that depends only on L, such that for all $x \in \Sigma^*$ and for all n, $C(\chi_{1:n}) \leq \log n + c_L$.*

Proof. (i) $\rightarrow$ (ii): By similar proof as the KC Regularity lemma.

(ii) $\rightarrow$ (iii): obvious.

(iii) $\rightarrow$ (iv): obvious.

(iv) $\rightarrow$ (i): shown in the following claim.

CLAIM 3.8. For each constant c there are only finitely many one-way infinite binary strings ω such that for all n, $C(\omega_{1:n}) \leq \log n + c$.

Proof. The claim is a weaker version of Theorem 6 in [2]. It turns out that the weaker version admits a simpler proof. To make the treatment self-contained, we present this new proof in the Appendix. □

By (iv) and the claim, there are only finitely many distinct χ's associated with the x's in Σ^*. Define the right-invariant equivalence relation $\sim$ by $x \sim x'$ if $\chi = \chi'$. This relation induces a partition of Σ^* in equivalence classes $[x] = \{y : y \sim x\}$. Since there is a one-one correspondence between the $[x]$'s and the χ's, and there are only finitely many distinct χ's, there are also only finitely many $[x]$'s, which implies that L is regular by the Myhill–Nerode theorem. □

REMARK 1. The KC Regularity lemma may be viewed as a corollary of the theorem. If L is regular, then clearly L_x is regular, and it follows immediately that there are only finitely many associated χ's, and each can be specified in at most c bits, where c is a constant depending only on L (and enumerator ϕ). If y is, say, the nth string in L_x, then we can specify y as the string corresponding to the nth '1' in χ, using only $C(n) + O(1)$ bits to specify y. Hence $C(y) \leq C(n) + O(1)$. Without loss of generality, we need to assume that the nth string enumerated in L_x in the KC-regularity Lemma is the string corresponding to the nth '1' in χ by the enumeration in the Theorem, or that there is a recursive mapping between the two.

REMARK 2. If L is nonregular, then there are infinitely many $x \in \Sigma^*$ with distinct equivalence classes $[x]$, each of which has its own distinct associated characteristic sequence χ. It is easy to see, for each automaton (finite or infinite) and for each χ associated with an equivalence class $[x]$, we have

$$C(\chi_{1:n}|n) \to \inf\{C(y) : y \in [x]\} + O(1),$$

for $n \to \infty$. The difference between finite and infinite automata is precisely expressed in the fact that only in the first case does there exist an a priori constant which bounds the left-hand term for all χ.

We show how to prove positive results with the KC characterization theorem. (Examples of negative results were given in the preceding section.)

EXAMPLE 7. Prove that $L = \Sigma^*$ is regular. There exists a constant c such that for each x the associated characteristic sequence is $\chi = 1, 1, \ldots$, with $C(\chi_{1:n}|n) \leq c$. Therefore, L is regular by the KC characterization theorem.

EXAMPLE 8. Prove that $L = \{x : x$ is accepted by a 2-way dfa$\}$ is regular. There exists a constant c such that for each x we have $C(\chi_{1:n}|n) \leq c$. Therefore, L is regular by the KC Characterization theorem.

4. Deterministic context-free languages. We present a Kolmogorov complexity based criterion to show that certain languages are not dcfl. In particular, it can be used to demonstrate the existence of witness languages in the difference of the family of context-free languages (cfls) and deterministic context-free languages (dcfls). Languages in this difference are the most difficult to identify; other non-dcfl are also non-cfl and in those cases we can often use the pumping lemma for context-free languages. The new method compares favorably with other known related techniques (mentioned in the Introduction) by being simpler, easier to apply, and apparently more powerful (because it works on a superset of examples). Yet our primary goal is to demonstrate the usefulness of Kolmogorov complexity in this matter.

A language is a dcfl iff it is accepted by a deterministic pushdown automaton (dpda).

Intuitively, Lemma 4.2 tries to capture the following. Suppose a dpda accepts $L = \{0^n 1^n 2^n : n \geq 1\}$. Then the dpda needs to first store a representation of the all 0 part, and then retrieve it to check against the all 1 part. But after that check, it seems inevitable that it has discarded the relevant information about n, and cannot use this information again to check against the all 2 part. That is, the complexity of the all 2 part should be $C(n) = O(1)$, which yields a contradiction for large n.

DEFINITION 4.1. *A one-way infinite string $\omega = \omega_1\omega_2 \ldots$ over Σ is* recursive *if there is a total recursive function $f : \mathcal{N} \to \Sigma$ such that $\omega_i = f(i)$ for all $i \geq 1$.*

LEMMA 4.2 (KC-DCFL). *Let $L \subseteq \Sigma^*$ be recognized by a deterministic pushdown machine M and let c be a constant. Let $\omega = \omega_1\omega_2 \ldots$ be a recursive sequence over Σ which can be described in c bits. Let $x, y \in \Sigma^*$ with $C(x, y) < c$ and let $\zeta = \ldots\zeta_2\zeta_1$ be a (reversed) recursive sequence over Σ of the form $\ldots yyx$. Let $n, m \in \mathcal{N}$ and $w \in \Sigma^*$ be such that items* (i)–(iii) *below are satisfied.*

(i) *For each i $(1 \leq i \leq n)$, given M's state and pushdown store contents after processing input $\zeta_m \ldots \zeta_1\omega_1 \ldots \omega_i$, a description of ω, and an additional description of at most c bits, we can reconstruct n by running M and observing only acceptance or rejection.*

(ii) *Given pushdown store contents after processing input $\zeta_m \ldots \zeta_1\omega_1 \ldots \omega_n$ and M's state, we can reconstruct w from an additional description of at most c bits.*

(iii) $C(\omega_1 \ldots \omega_n) \geq 2 \log\log m$.

Then there is a constant c' depending only on L and c such that $C(w) \leq c'$.

Proof. Let L be accepted by M with input head h_r. Assume that m, n, w satisfy the conditions in the statement of the lemma. For convenience we write

$$u = \zeta_m \ldots \zeta_1, \qquad v = \omega_1 \ldots \omega_n.$$

For each input $z \in \Sigma^*$, we denote with $c(z)$ the pushdown store contents at the time h_r has read all of z, and moves to the right adjacent input symbol. Consider the computation of M on input uv from the time when h_r reaches the end of u. There are two cases, which follow.

Case 1. There is a constant c_1 such that for infinitely many pairs m, n that satisfy the statement of the lemma, if h_r continues and reaches the end of v, then all of the original $c(u)$ has been popped except at most the bottom c_1 bits.

That is, machine M decreases its pushdown store from size $l(c(u))$ to size c_1 during the processing of v. The first time this occurs, let v' be the processed initial segment of v, and v'' the unprocessed suffix (so that $v = v'v''$) and let M be in state q. We can describe w by the following items:[2]

- A self-delimiting description of M (including Σ) and this discussion in $O(1)$ bits.
- A self-delimiting description of ω in $(1 + \epsilon)c$ bits.
- A description of $c(uv')$ and q in $c_1 \log |\Sigma| + O(1)$ bits.
- The "additional description" mentioned in item (i) of the statement of the lemma in self-delimiting format, using at most $(1 + \epsilon)c$ bits. Denote it by p.
- The "additional" description mentioned in item (ii) of the statement of the lemma in self-delimiting format, using at most $(1 + \epsilon)c$ bits. Denote it by r.

By item (i) in the statement of the lemma we can reconstruct v'' from M in state q and with pushdown store contents $c(uv')$, and ω, using description p. Subsequently, by starting M in state q with pushdown store contents $c(uv')$, we process v''. At the end of the computation we have obtained M's state and pushdown store contents after processing uv. According to item (ii) in the statement of the lemma, together with description r, we can now reconstruct w. Since $C(w)$ is at most the length of this description,

$$C(w) \leq 4c + c_1 \log |\Sigma| + O(1).$$

Setting $c' := 4c + c_1 \log |\Sigma| + O(1)$ satisfies the lemma.

Case 2. By way of contradiction, assume that Case 1 does not hold. That is, for each constant c_1 all but finitely many pairs m, n that satisfy the conditions in the lemma cause M not to decrease its stack height below c_1 during the processing of the v part of input uv.

Fix some constant c_1. Set m, n so that they satisfy the statement of the lemma, and to be as long as required to validate the argument below. Choose u' as a suffix of $yy \ldots yx$ with $l(u') > 2^m$ and

$$C(l(u')) < \log \log m. \tag{2}$$

That is, $l(u')$ is much larger than $l(u)$ $(= m)$ and much more regular. A moment's reflection learns that we can always choose such a u'.

CLAIM 4.3. For large enough m there exists a u' as above, such that M starts in the same state and accesses the same top $l(c(u)) - c_1$ elements of its stack during the processing of the v parts of both inputs uv and $u'v$.

Proof. By assumption, M does not read below the bottom c_1 symbols of $c(u)$ while processing the v part of input uv.

We argue that one can choose u' such that the top segment of $c(u')$ is precisely the same as the top segment of $c(u)$ above the bottom c_1 symbol for large enough $l(u), l(u')$.

To see this we examine the initial computation of M on u. Since M is deterministic, it must either cycle through a sequence of pushdown store contents, or increase its pushdown store with repetitions on long enough u (and u'). Namely, let a triple (q, i, s) mean that M is in state q, has top pushdown store symbol s, and h_r is at ith bit of some y. Consider only the triples (q, i, s) at the steps where M will never go below the current top pushdown store level

[2]Since we need to glue together different binary items in the encoding, and in a way so that we can effectively separate them again, like $\langle x, y \rangle = x'y$, we count $C(x) + 2 \log C(x) + 1$ bits for a self-delimited encoding $x' = 1^{l(l(x))}0l(x)x$ of x. We only need to give self-delimiting forms for all but one constituent description item.

again while reading u. (That is, s will not be popped before going into v.) There are precisely $l(c(u))$ such triples. Because the input is repetitious and M is deterministic, some triple must start to repeat within a constant number of steps and with a constant interval (in height of M's pushdown store) after M starts reading y's. It is easy to show that within a repeating interval only a constant number of y's are read.

The pushdown store does not cycle through an a priori bounded set of pushdown store contents, since this would mean that there is a constant c_1 such that the processing by M of any suffix of $yy \ldots yx$ does not increase the stack height above c_1. This situation reduces to Case 1 with $v = \epsilon$.

Therefore, the pushdown store contents grow repetitiously and unboundedly. Since the repeating cycle starts in the pushdown store after a constant number of symbols, and its size is constant in number of y's, we can adjust u' so that M starts in the same state and reads the same top segments of $c(u)$ and $c(u')$ in the v parts of its computations on uv and $u'v$. This proves the claim. □

The following items form a description from which we can reconstruct v.

- This discussion and a description of M in $O(1)$ bits.
- A self-delimiting description of the recursive sequence ω of which v is an initial segment in $(1 + \epsilon)c$ bits.
- A self-delimiting description of the pair $\langle x, y \rangle$ in $(1 + \epsilon)c$ bits.
- A self-delimiting description of $l(u')$ in $(1 + \epsilon)C(l(u'))$ bits.
- A program p to reconstruct v given ω and M's state and pushdown store contents after processing u. By item (i) of the statement of the lemma, $l(p) \leq c$. Therefore, a self-delimiting description of p takes at most $(1 + \epsilon)c$ bits.

The following procedure reconstructs v from this information. By using the description of M and u' we construct the state $q_{u'}$ and pushdown store contents $c(u')$ of M after processing u'. By Claim 4.3, the state q_u of M after processing u satisfies $q_u = q_{u'}$ and the top $l(c(u)) - c_1$ elements of $c(u)$ and $c(u')$ are the same. Run M on input ω starting in state $q_{u'}$ and with stack contents $c(u')$. By assumption, no more than $l(c(u)) - c_1$ elements of $c(u')$ get popped before we have processed $\omega_1 \ldots \omega_n$. By just looking at the consecutive states of M in this computation, and using program p, we can find n according to item (i) in the statement of the lemma. To reconstruct v requires by definition at least $C(v)$ bits. Therefore,

$$\begin{aligned} C(v) &\leq (1 + \epsilon)C(l(u')) + 4c + O(1) \\ &\leq (1 + \epsilon) \log \log m + 4c + O(1), \end{aligned}$$

where the last inequality follows by equation (2). But this contradicts item (iii) in the statement of the lemma for large enough m. □

Items (i)–(iii) in the KC-DCFL lemma can be considerably weakened, but the presented version gives the essential idea and power: it suffices for many examples. A more restricted, but easier, version is the following.

COROLLARY 4.4. *Let $L \subseteq \Sigma^*$ be a dcfl and let c be a constant. Let x and y be fixed finite words over Σ and let ω be a recursive sequence over Σ. Let u be a suffix of $yy \ldots yx$, let v be a prefix of ω, and let $w \in \Sigma^*$ such that*

(i) *v can be described in c bits given L_u in lexicographical order;*

(ii) *w can be described in c bits given L_{uv} in lexicographical order; and*

(iii) $C(v) \geq 2 \log \log l(u)$.

Then there is a constant c' depending only on L, c, x, y, ω such that $C(w) \leq c'$.

All the following context-free languages were proved to be not dcfl only with great effort before [6], [4], [24]. Our new proofs are more direct and intuitive. Basically, if v is the first word in L_u, then processing the v part of input uv must have already used up the information

of u. But if there is not much information left on the pushdown store, then the first word w in L_{uv} cannot have high Kolmogorov complexity.

EXAMPLE 9 (Exercise 10.5 (a)** in [6]). Prove $L = \{x : x = x^R, x \in \{0, 1\}^*\}$ is not dcfl. Suppose the contrary. Set $u = 0^n1$ and $v = 0^n$, $C(n) \geq \log n$, which satisfies item (iii) of the lemma. Since v is lexicographically the first word in L_u, item (i) of the lemma is satisfied. The lexicographically first nonempty word in L_{uv} is 10^n, and so we can set $w = 10^n$ which satisfies item (ii) of the lemma. But now we have $C(w) = \Omega(\log n)$, which contradicts the KC-DCFL lemma and its corollary.

Approximately the same proof shows that the context-free language $\{xx^R : x \in \Sigma^*\}$ and the context-sensitive language $\{xx : x \in \Sigma^*\}$ are not deterministic context-free languages.

EXAMPLE 10 (Exercise 10.5 (b)** in [6] and Example 1 in [24]). Prove $\{0^n1^m : m = n, 2n\}$ is not dcfl. Suppose the contrary. Let $u = 0^n$ and $v = 1^n$, where $C(n) \geq \log n$. Then v is the lexicographically first word in L_u. The lexicographically first nonempty word in L_{uv} is 1^n. Set $w = 1^n$, and $C(w) = \Omega(\log n)$, contradicting the KC-DCFL lemma and its corollary.

EXAMPLE 11 (Example 2 in [24]). Prove $L = \{xy : l(x) = l(y), y$ contains a "1," $x, y \in \{0, 1\}^*\}$ is not dcfl. Suppose the contrary. Set $u = 0^n1$ where $l(u)$ is even. Then $v = 0^{n+1}$ is lexicographically the first even length word not in L_u. With $C(n) \geq \log n$, this satisfies items (i) and (iii) of the lemma. Choosing $w = 10^{2n+3}$, the lexicographically first even length word not in L_{uv} starting with a "1", satisfies item (ii). But $C(w) = \Omega(\log n)$, which contradicts the KC-DCFL lemma and its corollary.

EXAMPLE 12. Prove $L = \{0^i1^j2^k : i, j, k \geq 0, i = j$ or $j = k\}$ is not dcfl. Suppose the contrary. Let $u = 0^n$ and $v = 1^n$, where $C(n) \geq \log n$, satisfying item (iii) of the lemma. Then, v is lexicographically the first word in L_u, satisfying item (i). The lexicographic first word in $L_{uv} \cap \{1\}\{2\}^*$ is 12^{n+1}. Therefore, we can set $w = 12^{n+1}$ and satisfy item (ii). Then $C(w) = \Omega(\log n)$, contradicting the KC-DCFL lemma and its corollary.

EXAMPLE 13 (pattern-matching). The KC-DCFL lemma and its corollary can be used in a tricky manner. We prove $\{x\#yx^Rz : x, y, z \in \{0, 1\}^*\}$ is not dcfl. Suppose the contrary. Let $u = 1^n\#$ and $v = 1^{n-1}0$, where $C(n) \geq \log n$, which satisfies item (iii) of the lemma. Since $v' = 1^n$ is the lexicographically first word in L_u, the choice of v satisfies item (i) of the lemma. (We can reconstruct v from v' by flipping the last bit of v' from 1 to 0.) Then $w = 1^n$ is lexicographically the first word in L_{uv}, to satisfy item (ii). Since $C(w) = \Omega(\log n)$, this contradicts the KC-DCFL lemma and its corollary.

5. Recursive, recursively enumerable, and beyond. It is immediately obvious how to characterize recursive languages in terms of Kolmogorov complexity. If $L \subseteq \Sigma^*$, and $\Sigma^* = \{v_1, v_2, \ldots\}$ is effectively ordered, then we define the characteristic sequence $\lambda = \lambda_1, \lambda_2, \ldots$ of L by $\lambda_i = 1$ if $v_i \in L$ and $\lambda_i = 0$ otherwise. In terms of the earlier developed terminology, if A is the automaton accepting L, then λ is the characteristic sequence associated with the equivalence class $[\epsilon]$. Recall Definition 4.1 of a recursive sequence. A set $L \in \Sigma^*$ is recursive iff its characteristic sequence λ is a recursive sequence. The next theorem then follows from the definitions and the first paragraph of the Appendix.

THEOREM 5.1 (recursive KC characterization). *A set $L \in \Sigma^*$ is recursive iff there exists a constant c_L (depending only on L) such that, for all n, $C(\lambda_{1:n}|n) < c_L$.*

L is r.e. (recursively enumerable) if the set $\{n : \lambda_n = 1\}$ is r.e. In terms of Kolmogorov complexity, the following theorem gives not only a qualitative but even a quantitative difference between recursive and r.e. languages. The following theorem is due to Barzdin' [1], [13].

THEOREM 5.2 (KC r.e.). (i) *If L is r.e., then there is a constant c_L (depending only on L), such that for all n, $C(\lambda_{1:n}|n) \leq \log n + c_L$.*

(ii) *There exists an r.e. set L such that $C(\lambda_{1:n}) \geq \log n$, for all n.*

Note that, with L as in item (ii), the set $\Sigma^* - L$ (which is possibly non-r.e.) also satisfies item (i). Therefore, item (i) is not a Kolmogorov complexity characterization of the r.e. sets.

EXAMPLE 14. Consider the standard enumeration of Turing machines. Define $k = k_1 k_2 \ldots$ by $k_i = 1$ if the ith Turing machine started on its ith program halts ($\phi_i(i) < \infty$), and $k_i = 0$ otherwise. Let A be the language such that k is its characteristic sequence. Clearly, A is an r.e. set. In [1] it is shown that $C(k_{1:n}) \geq \log n$ for all n.

EXAMPLE 15. Let k be as in the previous example. Define a one-way infinite binary sequence h by

$$h = k_1 0^2 k_2 0^{2^2} \ldots k_i 0^{2^i} k_{i+1} \ldots .$$

Then, $C(h_{1:n}) = O(C(n)) + \Theta(\log \log n)$. Therefore, if h is the characteristic sequence of a set B, then B is not recursive, but more "sparsely" nonrecursive than A is.

EXAMPLE 16. The probability that the optimal universal Turing machine U halts on self-delimiting binary input p, randomly supplied by tosses of a fair coin, is Ω, $0 < \Omega < 1$. Let the binary representation of Ω be $0.\Omega_1\Omega_2 \ldots$ Let Σ be a finite nonempty alphabet, and $v_1, v_2, \ldots$ an effective enumeration without repetitions of Σ^*. Define $L \subseteq \Sigma^*$ such that $v_i \in L$ iff $\Omega_i = 1$. It can be shown (see, for example, [12]) that the sequence $\Omega_1, \Omega_2, \ldots$ satisfies

$$C(\Omega_{1:n}|n) \geq n - \log n - 2 \log \log n - O(1),$$

for all but finitely many n.

Hence neither L nor $\Sigma^* - L$ is r.e. It is not difficult to see that $L \in \Delta_2 - (\Sigma_1 \cup \Pi_1)$, in the arithmetic hierarchy (that is, L is not recursively enumerable) [22], [23].

6. Questions for future research. (a) It is not difficult to give a direct KC analogue of the $uvwxy$ pumping lemma (as Tao Jiang pointed out to us). Just like the pumping lemma, this will show that $\{a^n b^n c^n : n \geq 1\}$, $\{xx : x \in \Sigma^*\}$, $\{a^p : p \text{ is prime}\}$, and so on, are not cfl. Clearly, this hasn't yet captured the Kolmogorov complexity heart of cfl. In general, can we find a CFL-KC characterization?

(b) What about ambiguous context-free languages?

(c) What about context-sensitive languages and deterministic context-sensitive languages?

Appendix: Proof of Claim 3.8. A *recursive real* is a real number whose binary expansion is recursive in the sense of Definition 4.1. The following result is demonstrated in [14] and attributed to A.R. Meyer. For each constant c there are only finitely many $\omega \in \{0, 1\}^\infty$ with $C(\omega_{1:n}|n) \leq c$ for all n. Moreover, each such ω is a recursive real.

In [2] this is strengthened to a version with $C(\omega_{1:n}) \leq C(n) + c$, and strengthened again to a version with $C(\omega_{1:n}) \leq \log n + c$. Claim 3.8 is weaker than the latter version by not requiring the ω's to be recursive reals. For completeness sake, we present a new direct proof of Claim 3.8 avoiding the notion of recursive reals.

Recall our convention of identifying integer x with the xth binary sequence in lexicographical order of $\{0, 1\}^*$ as in (1).

Proof of Claim 3.8. Let c be a positive constant, and let

$$\begin{aligned} A_n &= \{x \in \{0, 1\}^n : C(x) \leq \log n + c\}, \\ A &= \{\omega \in \{0, 1\}^\infty : \forall_{n \in \mathcal{N}}[C(\omega_{1:n}) \leq \log n + c]\} . \end{aligned} \tag{3}$$

If the cardinality $d(A_n)$ of A_n dips below a fixed constant c', for infinitely many n, then c' is an upper bound on $d(A)$. This is because it is an upper bound on the cardinality of the set of prefixes of length n of the elements in A, for *all* n.

Fix any $l \in \mathcal{N}$. Choose a binary string y of length $2l + c + 1$ that satisfies

$$C(y) \geq 2l + c + 1. \tag{4}$$

Choose i maximum such that for division of y in $y = mn$ with $l(m) = i$ we have

$$m \leq d(A_n). \tag{5}$$

(This holds at least for $i = 0 = m$.) Define similarly a division $y = sr$ with $l(s) = i + 1$. By maximality of i, we have $s > d(A_r)$. From the easily proven $s \leq 2m + 1$, it then follows that

$$d(A_r) \leq 2m. \tag{6}$$

We prove $l(r) \geq l$. Since by (5) and (3) we have

$$m \leq d(A_n) \leq 2^c n,$$

it follows that $l(m) \leq l(n) + c$. Therefore,

$$2l + c + 1 = l(y) = l(n) + l(m) \leq 2l(n) + c,$$

which implies that $l(n) > l$. Consequently, $l(r) = l(n) - 1 \geq l$.

We prove $d(A_r) = O(1)$. By dovetailing the computations of the reference universal Turing machine U for all programs p with $l(p) \leq \log n + c$, we can enumerate all elements of A_n. We can reconstruct y from the mth element, say y_0, of this enumeration. Namely, from y_0 we reconstruct n since $l(y_0) = n$, and we obtain m by enumerating A_n until y_0 is generated. By concatenation we obtain $y = mn$. Therefore,

$$C(y) \leq C(y_0) + O(1) \leq \log n + c + O(1). \tag{7}$$

From (4), we have

$$C(y) \geq \log n + \log m. \tag{8}$$

Combining (7) and (8), it follows that $\log m \leq c + O(1)$. Therefore, by (6),

$$d(A_r) \leq 2^{c+O(1)}.$$

Here, c is a fixed constant independent of n and m. Since $l(r) \geq l$ and we choose l arbitrarily, $d(A_r) \leq c_0$ for a fixed constant c_0 and infinitely many r, which implies $d(A) \leq c_0$, and hence the claim. □

We avoided establishing, as in the cited references, that the elements of A defined in (3) are recursive reals. The resulting proof is simpler, and sufficient for our purpose, since we only need to establish the finiteness of A.

REMARK 3. The difficult part of the Regular KC Characterization theorem above consists in proving that the KC Regularity lemma is exhaustive, i.e., can be used to prove the nonregularity of all nonregular languages. Let us look a little more closely at the set of sequences defined in item (iii) of the KC Characterization theorem. The set of sequences A of (3) is a superset of the set of characteristic sequences associated with L. According to the proof in the cited references, this set A contains finitely many *recursive* sequences (computable by Turing machines). The subset of A consisting of the characteristic sequences associated with L, satisfies much more stringent computational requirements, since it can be computed using only the finite automaton recognizing L. If we replace the plain Kolmogorov complexity in the statement of the theorem by the so-called "prefix complexity" variant K, then the equivalent set of A in (3) is

$$\{\omega \in \{0, 1\}^\infty : \forall_{n \in \mathcal{N}}[K(\omega_{1:n}) \leq K(n) + c]\},$$

which is finite [12, Exercise 3.24] and contains nonrecursive sequences by a result of Solovay [20].

Acknowledgments. We thank Peter van Emde Boas, Theo Jansen, Tao Jiang for reading the manuscript and commenting on it, and the anonymous referees for extensive comments and suggestions for improvements. John Tromp improved the proof of Claim 3.8.

REFERENCES

[1] Y.M. BARZDIN', *Complexity of programs to determine whether natural numbers not greater than n belong to a recursively enumerable set*, Soviet Math. Dokl., 9 (1968), pp. 1251–1254.

[2] G.J. CHAITIN, *Information-theoretic characterizations of recursive infinite strings*, Theoret. Comput. Sci., 2 (1976), pp. 45–48.

[3] A. EHRENFEUCHT, R. PARIKH, AND G. ROZENBERG, *Pumping lemmas for regular sets*, SIAM J. Comput., 10 (1981), pp. 536–541.

[4] M.A. HARRISON, *Introduction to Formal Language Theory*, Addison-Wesley, Reading, MA, 1978.

[5] D.R. HEATH-BROWN AND H. IWANIEC, *The difference between consecutive primes*, Invent. Math., 55 (1979), pp. 49–69.

[6] J.E. HOPCROFT AND J.D. ULLMAN, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[7] J. JAFFE, *A necessary and sufficient pumping lemma for regular languages*, SIGACT News, 10 (1978), pp. 48–49.

[8] T. JIANG AND M. LI, *k one-way heads cannot do string matching*, Proc. 25th ACM Symp. Theory of Computing, ACM Press, 1993, pp. 62–70.

[9] T. JIANG, J.I. SEIFERAS, AND P. VITÁNYI, *Two heads are better than two tapes*, Proc. 26th ACM Symp. Theory of Computing, ACM Press, 1994, pp. 668–675.

[10] A.N. KOLMOGOROV, *Three approaches to the quantitative definition of information*, Problems Inform. Transmission, 1 (1965), pp. 1–7.

[11] M. LI AND P.M.B. VITÁNYI, *Tape versus queue and stacks: The lower bounds*, Inform. and Comput., 78 (1988), pp. 56–85.

[12] ———, *An Introduction to Kolmogorov Complexity and its Applications*, Springer-Verlag, New York, 1993.

[13] D.W. LOVELAND, *On minimal-program complexity measures*, in Proc. (1st) ACM Symp. Theory of Computing, ACM Press, pages 61–65, 1969.

[14] ———, *A variant of the Kolmogorov concept of complexity*, Inform. and Control, 15 (1969), pp. 510–526.

[15] W. MAASS, *Combinatorial lower bound arguments for deterministic and nondeterministic Turing machines*, Trans. Amer. Math. Soc., 292 (1985), pp. 675–693.

[16] P. MARTIN-LÖF, *The definition of random sequences*, Inform. and Control, 9 (1966), pp. 602–619.

[17] ———, *Complexity oscillations in infinite binary sequences*, Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete, 19 (1971), pp. 225–230.

[18] W. J. PAUL, *Kolmogorov's complexity and lower bounds*, in L. Budach, ed., Proc. 2nd International Conference on Fundamentals of Computation Theory, pp. 325–334, Akademie Verlag, Berlin, 1979.

[19] W.J. PAUL, J.I. SEIFERAS, AND J. SIMON, *An information theoretic approach to time bounds for on-line computation*, J. Comput. System Sci., 23 (1981), pp. 108–126.

[20] R. SOLOVAY, lecture notes, unpublished, University of California at Los Angeles, 1975.

[21] D. STANAT0 AND S. WEISS, *A pumping theorem for regular languages*, SIGACT News, 14 (1982), pp. 36–37.

[22] M. VAN LAMBALGEN, *Random Sequences*, Ph.D. thesis, Universiteit van Amsterdam, Amsterdam, 1987.

[23] ———, *Algorithmic information theory*, J. Symbolic Logic, 54 (1989), pp. 1389–1400.

[24] S. YU, *A pumping lemma for deterministic context-free languages*, Inform. Process. Lett., 31 (1989), pp. 47–51.

SIAM J. COMPUT.
Vol. 24, No. 3, pp. 411–439, June 1995

$O(M \cdot N)$ ALGORITHMS FOR THE RECOGNITION AND ISOMORPHISM PROBLEMS ON CIRCULAR-ARC GRAPHS*

WEN-LIAN HSU†

Abstract. Circular-arc graphs have a rich combinatorial structure. The circular endpoint sequence of arcs in a model for a circular-arc graph is usually far from unique. We present a natural restriction on these models to make it meaningful to define the unique representations for circular-arc graphs. We characterize those circular-arc graphs which have unique restricted models and give an $O(m \cdot n)$ algorithm for recognizing circular-arc graphs. We think a more careful implementation could reduce the complexity to $O(n^2)$. Our approach is to reduce the recognition problem of circular-arc graphs to that of circle graphs. This approach has the following advantages: it is conceptually simpler than Tucker's $O(n^3)$ recognition algorithm; it exploits the similarity between circle graphs and circular-arc graphs in a natural fashion; it yields an isomorphism algorithm. A main contribution of this result is an illustration of the transformed decomposition technique. The decomposition tree developed for circular-arc graphs generalizes the concept of the PQ-tree, which is a data structure that keeps track of all possible interval representations of a given interval graph. As a consequence, our approach also yields an $O(m \cdot n)$ isomorphism algorithm for circle graphs.

Key words. circular-arc graph, circle graph, graph decomposition, complexity

AMS subject classifications. 68Q25, 68R05, 68R10

1. Introduction.

Intersection graphs have recently received much attention in the study of algorithmic graph theory and their applications (see e.g., [1], [8], [9], [12], [15], [20]). Well-known special classes of intersection graphs include interval graphs, chordal graphs, circular-arc graphs, permutation graphs, circle graphs, and so on. Various optimization problems (e.g., maximum clique, maximum independent set, and minimum coloring problems) on these graphs have also been studied extensively [6], [7], [10], [11], [13], [14], [21]. We shall denote a graph G by a pair (V, E), where V denotes the finite vertex set of G and E denotes a set of edges connecting vertices of G. Let $n = |V|$ and $m = |E|$.

Circular-arc graphs have applications in genetic research [18], traffic control [19], and computer compiler design [21]. Tucker [22] has given an $O(n^3)$ algorithm for recognizing circular-arc graphs. However, the algorithm and its proof are involved. Based on Tucker's algorithm, Wu [23] wrote a dissertation on the isomorphism problem of circular-arc graphs (but it never appeared in journal form). Spinrad [17] simplified Tucker's algorithm for the case that the given graph can be covered by two cliques.

In this paper we present $O(m \cdot n)$ algorithms for both the recognition and isomorphism problems on circular-arc graphs. We think a more careful implementation could reduce the complexity to $O(n^2)$. Our approach is to reduce the circular-arc graph recognition problem to the circle graph recognition problem. We believe this approach has the following advantages: (1) it is conceptually simple, (2) it exploits the similarity between circle graphs and circular-arc graphs in a natural fashion, and (3) it yields an isomorphism algorithm for circular-arc graphs as a by-product.

A main contribution of our result is an illustration of the "transformed decomposition" technique. A common drawback of ordinary decomposition techniques is that they do not seem to yield useful reduction for the specific class of graphs of interest. Our example illustrates how one might be able to modify the given class to another one for which decomposition produces fruitful reduction. In the study of interval graphs (a subclass of circular-arc graphs), a new data structure, PQ-trees [1], was created to keep track of all possible interval representations of a given interval graph. The decomposition tree discovered in this paper generalizes the

*Received by the editors March 1, 1993; accepted for publication (in revised form) May 23, 1994. This research was supported in part by the National Science Council of the Republic of China.

†Institute of Information Science, Academia Sinica, Taipei, Taiwan, Republic of China.

concept of PQ-trees and keeps track of all possible representations of a given circular-arc graph. We also give characterizations of the class of uniquely representable interval graphs as well as circular-arc graphs. Finally, for the two widely used techniques, substitution and join decomposition, our result illustrates a case where they can be blended together harmoniously to complement each other.

The difficulty in dealing with circular-arc graphs lies largely in the fact that even a simple circular-arc graph (or interval graph) can have an exponential number of arc representations. We propose a notion of *normalized representations* with the property that the relative endpoint positions of any two arcs in such representations are prescribed by the adjacency relationships of their corresponding vertices in G. By restricting ourselves to these special representations, we can eliminate trivial variations. Through graph decomposition, we provide a characterization of those circular-arc graphs that possess "unique" normalized representations.

We now define some notations used in the paper. Let F be a family of nonempty sets. The *intersection graph* of F is obtained by representing each set in F by a vertex and connecting two vertices by an edge if and only if their corresponding sets have a nonempty intersection. When F is a family of intervals on the real line, the intersection graph is called an *interval graph*. When F is a family of arcs (resp., chords) on a circle, the intersection graph is called a *circular-arc graph* (resp., *circle graph*). A permutation graph is a circle graph in which the circle can be divided into two halves, say part A and part B, and each chord has one end in A and the other end in B. Each of these representations using intervals, arcs, or chords is called a *model* for the corresponding graph.

Let R be an arc model of a circular-arc graph G. Without loss of generality, assume all arc endpoints in R are distinct. Each point on the circle has a coordinate, denoted by p. Let the arcs of F be denoted as $1, 2, \ldots, n$. Since it is the relative positions (not the actual coordinates) of these endpoints that concern us in this paper, we shall label the endpoints as follows. For each arc i, traversing from a point of the circle not in i along the clockwise direction, the first endpoint of i encountered is called the *head* of i (labeled by $h(i)$), and the second endpoint of i encountered is called the *tail* of i (labeled by $t(i)$). Denote arc i by $(h(i), t(i))$. Associate with R a *circular label sequence* obtained from a clockwise traversal (see Fig. 1.1) of the circle starting with any endpoint label. In the special case of an interval graph, we associate with its interval model a *left–right label sequence*. Denote the coordinate of a label m by $p(m)$. For two points c and d on the circle, define *segment* (c, d) to be the part of the circle obtained by traversing from c along the clockwise direction to d. For convenience, an arc $(h(i), t(i))$ is also used to denote the segment $(p(h(i)), p(t(i)))$. From now on, we shall only consider the label of an endpoint rather than its actual coordinate.

Two models for a circular-arc graph (resp., interval) are said to be *equivalent* if a circular (resp., left–right) label sequence of one model can be obtained from that of the other through rotation or reflection (resp., through reflection). A circular-arc (resp., interval) graph G is said to have a *unique* circular-arc (resp., interval) model if all circular-arc (resp., interval) models for G are equivalent. It is easy to see that a given circular-arc graph can have an exponential number of nonequivalent models. For example, by permuting the head labels and the tail labels of n mutually overlapping intervals, one can obtain $(n!)^2/2$ different models. In the next section, we introduce the notion of normalized models that eliminates a large number of trivial model variations. Note that equivalent chord models for a circle graph have been defined analogously in [5].

The main technique used in our recognition algorithm is graph decomposition. This is a useful divide-and-conquer scheme for attacking many optimization and recognition problems. Gabor, Hsu, and Supowit [5] and, independently, Bouchet [2] used the join decomposition to

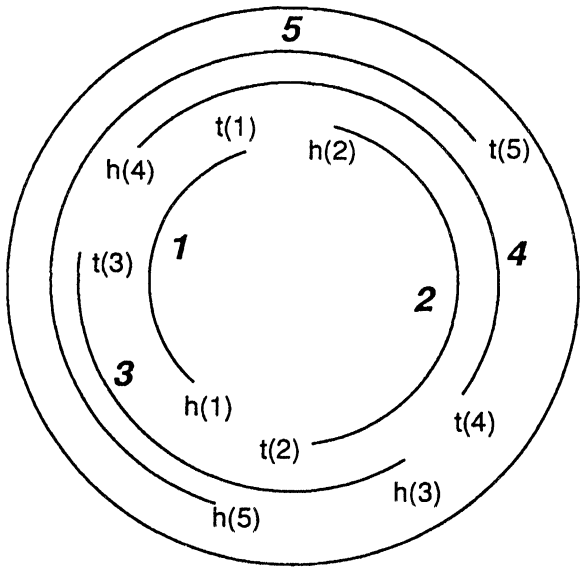


FIG. 1.1. *The circular label sequence of a circular-arc graph.*

efficiently recognize circle graphs. We shall adopt the same decomposition here in recognizing circular-arc graphs.

The idea behind these decompositions is discussed in greater detail in §3. In the next section we introduce the notion of normalized models for a circular-arc graph, which is used throughout the remainder of this paper. Section 4 describes uniquely representable interval graphs. Section 5 is devoted to the transformation of circular-arc graphs to circle graphs. The decomposition of circular-arc graphs is discussed in §§6 and 7. Section 8 discusses the time complexity of our algorithm. Finally, we show that our approach yields an isomorphism algorithm on circular-arc graphs in §9.

2. Normalized models for circular-arc graphs. We are interested in those circular-arc models in which the overlapping relationships (described below) of any two arcs are strictly dictated by their corresponding neighborhood structures in the graph (and therefore remain the same in all these representing models).

DEFINITION. *Given a circular-arc model R, there are four possible overlapping relationships on a pair of arcs* u_2 *and* u_2 *in D defined as follows (an example is shown in Fig.* 2.1):

(1) u_1 *is independent of* u_2 *if they do not overlap.*

(2) u_1 *is contained in (resp., contains)* u_2 *if segment* $(h(u_1), t(u_1))$ *is contained in (resp., contains) segment* $(h(u_2), t(u_2))$.

(3) u_1 *and* u_2 *strictly overlap if their four endpoints appear alternately on the circle.*

(4) u_1 *and* u_2 *cover the circle if the two segments* $(h(u_1), t(u_1))$ *and* $(h(u_2), t(u_2))$ *together cover the whole circle.*

For the special class of interval graphs, only the first three relationships are applicable when considering their interval representations.

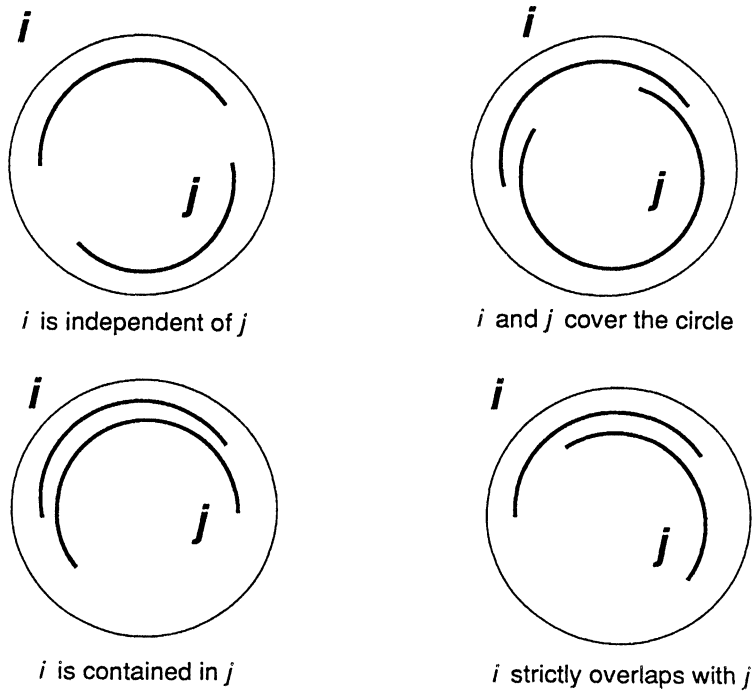


FIG. 2.1. *Four possible overlapping relationships on arcs u_1 and u_2.*

Denote by $N(v)$ (or $N_G(v)$) the set consisting of v and all vertices adjacent to v in G.

DEFINITION. *Two vertices v_1 and v_2 in a graph G are said to be*

(1) *independent if v_1 is not adjacent to v_2.*

(2) *strictly adjacent if v_1 is adjacent to v_2 but neither $N(v_1)$ nor $N(v_2)$ is contained in the other.*

(3) *strongly adjacent if v_1 and v_2 are strictly adjacent but every w in $V(G)\backslash N(v_1)$ satisfies that $N(w) \subsetneq N(v_2)$ and every w' in $V(G)\backslash N(v_2)$ satisfied that $N(w') \subseteq N(v_1)$.*

(4) *similar if $N(v_1)\backslash\{v_1\} = N(v_2)\backslash\{v_2\}$ (v_1, v_2 are said to form a similar pair).*

These relationships can be determined by finding the partial order (based on $\supseteq$) on the containment relationships of the $N(v)$'s for all v (details in §8). A vertex v in G is said to be a *D-vertex* (short for "dominating") if $N(v) = V(G)$. To avoid ambiguity, normalized models are defined only for those circular-arc graphs that contain neither similar pairs nor D-vertices.

DEFINITION. *Let G be a circular-arc graph containing neither similar pairs nor D-vertices. A circular-arc model R for G is said to be a normalized model (N-model) if every pair of arcs u_2 and u_2 in R and their corresponding vertices v_1 and v_2 in G satisfy the following conditions*:

(1) *u_1 is independent of $u_2 \Leftrightarrow v_1$ is not adjacent to v_2.*

(2) *u_1 is contained in $u_2 \Leftrightarrow N(v_1) \subseteq N(v_2)$.*

(3) *u_1 strictly overlaps with $u_2 \Leftrightarrow v_1$ and v_2 are strictly but not strongly adjacent.*

(4) *u_1 and u_2 cover the circle $\Leftrightarrow v_1$ and v_2 are strongly adjacent.*

For interval graphs, an interval N-model is defined with (3) *and* (4) *combined into*

(3′) *u_1 strictly overlaps with $u_2 \Leftrightarrow v_1$ and v_2 are strictly adjacent.*

Since the conditions on the vertex pair are mutually disjoint and collectively exhaustive, the notion "normalized model" is well defined. The overlapping relationships of all pairs of arcs in an N-model are completely determined by their corresponding vertex adjacency relationships.

Next, we describe an algorithm that transforms any circular-arc model of G into an N-model provided there are no similar pairs or D-vertices in G. Let R be a circular-arc model for G. It suffices to find a model R^* in which two arcs cross exactly when their corresponding vertices are strictly but not strongly adjacent, since this condition will force R^* to be an N-model.

DEFINITION. *A pair of arcs u_1 and u_2 are said to form a* "violation" *if they strictly overlap with each other in R but either their corresponding vertices v_1 and v_2 are strongly adjacent* (type I violation) *or they are not strictly adjacent* (type II violation).

We divide our algorithm into two parts. Define a *head* (or *tail*) *block* of a model R to be a set of maximal contiguous subsequence of head (or tail) labels in the circular label sequence of R. The term "block" refers to either a head block or a tail block. Given a block B, $NEXT(B)$ (resp., $PREV(B)$) refers to the neighboring block of B in the clockwise (resp., counterclockwise) direction. By inserting an endpoint into a block B, we mean inserting its label into the subsequence of B in arbitrary order. It is easy to check that, in both algorithms, reordering within a block does not change the arc overlapping relationships.

ALGORITHM I (ELIMINATING TYPE I PAIRS)

1. For each head $h(i)$, let $T(h(i))$ be the first tail block (encountered in a counterclockwise traversal from $h(i)$) that contains the tail of some arc not overlapping with arc i. Partition $T(h(i))$ into two blocks $T_1 = \{t(i_1), \ldots, t(i_r)\}$ and $T_2 = \{t(j_1), \ldots, t(j_s)\}$ where T_1 contains those tails whose corresponding arcs are not overlapping with i and $T_2 = T(h(i))\backslash T_1$. If $T_2 = \emptyset$, then insert $h(i)$ into $NEXT(T(h(i)))$. If $T_2 \neq \emptyset$, then insert a new head block $B' = \{h(i)\}$ in between T_1 and T_2.
2. Repeat the above procedure symmetrically for each tail $t(i)$ in the clockwise direction.

LEMMA 2.1. *Let R_1 be the model obtained at the end of Algorithm* I. *Then, R_1 does not contain any type* I *pair.*

Proof. Suppose otherwise. Let i_1, i_2 be a pair crossing in R_1 with their corresponding vertices v_1 and v_2 strongly adjacent in G. Without loss of generality, assume the circular label sequence for R_1 is $h(i_1)h(i_2)t(i_1)t(i_2)$. Let H_1 be the block containing $h(i_1)$ and T_2 the block containing $t(i_2)$.

By Algorithm I, there exists an arc j not overlapping with i_1 such that $t(j)$ is in $PREV(H_1)$ and another arc j' not overlapping with i_2 such that $h(j')$ is in $NEXT(T_2)$. Let the corresponding vertices of j and j' be w and w', respectively. Then, $w \notin N(v_1)$ and $w' \notin N(v_2)$. Since v_1 is strongly adjacent to v_2, we must have that $w \in N(v_2)$, $w' \in N(v_1)$. Then, $t(j)$ must be in arc j' and w is adjacent to w'. But then, we have $N(w) \not\subseteq N(v_2)$, contradictory to the strong adjacency of v_1 and v_2. □

ALGORITHM II (ELIMINATING TYPE II PAIRS)

1. For each head block $H = \{h(i_1), \ldots, h(i_r)\}$ in R_1, let the clockwise tail order of arcs $i_1, \ldots, i_r$ in R_1 be $i'_1, \ldots, i'_r$ (transversing from $h(i_r)$). Then order the head labels in H in the reverse order as $h(i'_r), \ldots, h(i'_1)$.
2. Reorder the tails in each tail block in reverse to the clockwise order of their heads.

Note that all arcs with their heads in the same block must have their corresponding tail labels in distinct tail blocks; otherwise, we would have similar pairs. Hence, the relative order of their tails remains the same throughout the algorithm and there is no need to reorder the same head block. The same holds for the tail blocks.

LEMMA 2.2. *Let R be the arc model obtained from R_1 at the end of Algorithm* II. *Then R is an N-model.*

Proof. Since we do not move any arc label out of its current block and R_1 does not contain any type I pair, R cannot contain any type I pair. Next, we show that R does not contain any

type II pair. Suppose otherwise. Let i_1, i_2 be a pair crossing in R_1 such that their corresponding vertices v_1 and v_2 satisfy $N(v_1) \subseteq N(v_2)$.

Without loss of generality, assume the circular label sequence for R is $h(i_1)h(i_2)t(i_1)t(i_2)$. $h(i_1)$ and $h(i_2)$ cannot belong to the same head block, otherwise their relative order would be $h(i_2)h(i_1)t(i_1)t(i_2)$ by Algorithm II. Hence, there must exist a tail $t(j)$ in segment $(h(i_1), h(i_2))$ such that arc j does not overlap with arc i_2, contradictory to the fact that $N(v_1) \subseteq N(v_2)$. □

COROLLARY 2.3. *Let G be a circular-arc graph containing neither similar pairs nor D-vertices. Then there exists an N-model for G. If, in particular, G is an interval graph, then there also exists an interval N-model for G.*

From here on, we assume the given graph does not contain similar pairs or D-vertices and consider only normalized models. To characterize graphs having unique N-models, we need to apply certain graph decompositions, which are discussed in the next section.

3. The substitution decomposition and the join decomposition. We shall consider two important graph decompositions in this section. The first such operation is substitution decomposition, also known as *modular decomposition.* We shall follow closely the notations of Spinrad [16], in which an $O(n^2)$ decomposition algorithm is given. Let V' be a subset of V. Denote by $G[V']$ the subgraph of G induced on V'.

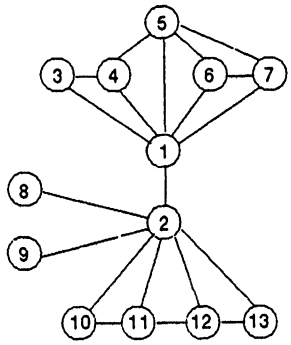
DEFINITION. *A graph G* contains a substitution *if there exists a partition of $V(G)$ into V_0, V_1, and V_2 with $|V_0| \geq 2$ and $|V_1 \cup V_2| \geq 1$ such that every vertex in V_0 is adjacent to every one in V_1 and no vertex in V_0 is adjacent to any one in V_2. Then, G is decomposable into $G[V_0]$ and the graph $G[V_1 \cup V_2]$ with an additional vertex v_0 connected to every vertex in V_1. Such a decomposition is called the* substitution decomposition *or* modular decomposition.

DEFINITION. *A* module *M is a subset of $V(G)$ such that between M and each vertex v in $V \setminus M$ there are either no edges or all possible edges. M is a* connected *module iff $G[M]$ is connected. M is* complement connected *iff the complement of $G[M]$ is connected. An unconnected module is called a* parallel *module. A connected module whose complement is unconnected is called a* series *module. A module that is both connected and complement connected is called a* neighborhood *module. A module M of size greater than 1 is nontrivial if $M \neq V(G)$. A graph G is s-inseparable if it does not contain any nontrivial module.*

By definition, every module is exactly one of these three types. A series or parallel module M can be decomposed into the connected components of $G[M]$ or $\overline{G}[M]$. A module M ($\neq N$) is a *maximal submodule* of a neighborhood module N if no other proper submodule of N contains M. A neighborhood module can be uniquely decomposed into its maximal submodules as its components. The modular decomposition starts with the module $V(G)$ and creates a corresponding root node in the decomposition tree labeled with S, P, or N, depending on the type of the module. Then it decomposes this module into components M_1, $M_2, \ldots, M_k$ using one of the above three rules. For each M_i, find its modular decomposition tree and make its root node a son of the root node representing $V(G)$. The resulting tree is called a *modular decomposition tree* of G. In [15], it was shown that there exists a unique modular decomposition tree for a graph G, namely, all modular decomposition trees for G are isomorphic. Fig. 3.1 gives a sample graph with its modular decomposition tree.

The *representative graph* of a module M in the decomposition tree is a graph induced by a subset consisting of a single vertex from each component submodule of M. We denote the representative graph of each module by G_i and attach it to the root of this module. The representative graph of a series module is a clique. The representative graph of a parallel module is an independent set. The representative graph of a neighborhood module is s-inseparable by our definition.

SUBSTITUTION DECOMPOSITION



The graph G

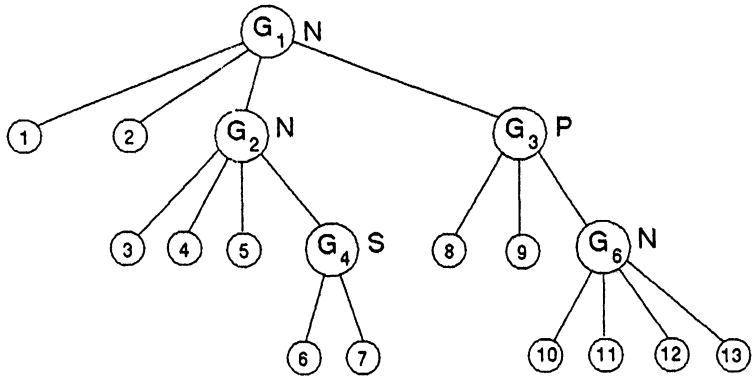


FIG. 3.1. *The substitution decomposition tree of a graph.*

DEFINITION. *The* s-inseparable components *of a graph G are defined to be either*

(1) *a single vertex if there is no neighborhood module in T, or*

(2) *the representative graphs of neighborhood modules in the decomposition tree T of G.*

The example in Fig. 3.1 has three neighborhood modules and each of their representative graphs is an induced path with four vertices.

Another decomposition is the *join decomposition* of Cunningham [3].

DEFINITION. *A graph G is said to have a* join *if* $V(G)$ *can be partitioned into* V_0, V_1, V_2, *and* V_3 *with* $|V_0 \cup V_1| \geq 2$ *and* $|V_2 \cup V_3| \geq 2$ *such that every possible edge exists between* V_1, V_2 *and no edge exists between* V_0, $V_2 \cup V_3$, *or between* $V_0 \cup V_1$, V_3. *In this case, we say G is decomposable into* H_1 *and* H_2 *where* H_1 *is the induced subgraph of* $G[V_0 \cup V_1]$ *with an extra vertex* v_1 *adjacent to every vertex in* V_1 *and* H_2 *is the induced subgraph* $G[V_2 \cup V_3]$ *with an extra vertex* v_2 *adjacent to every vertex in* V_2. *A graph G is said to be* j-inseparable *if it does not contain any join.*

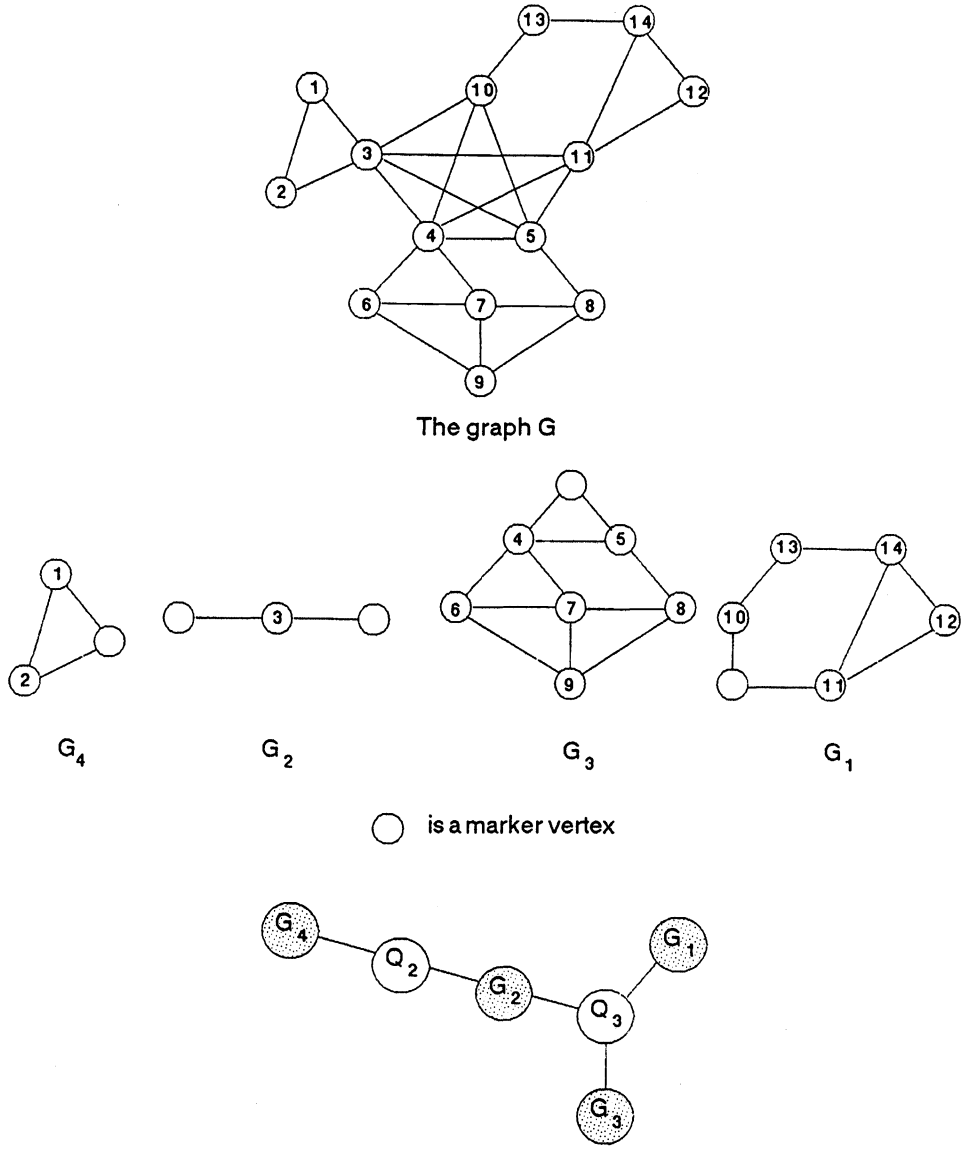


FIG. 3.2. *The join decomposition tree of a graph G.*

A j-inseparable graph with at least four vertices must be connected. Cunningham [3] gave an $O(n^3)$ algorithm to decompose a graph completely into j-inseparable components and showed that there is a unique join decomposition tree. (There is no complete description on how such a tree is formed; however, the reader familiar with the decomposition of a graph into 3-connected components should be able to figure out the construction.) Gabor, Hsu, and Supowit [5] later improved it to $O(m \cdot n)$ time (faster than [3] on sparse graphs). An example is shown in Fig. 3.2.

Our decomposition scheme on circular-arc graphs can be briefly described as follows. For the special class of interval graphs, we show that their s-inseparable components have

unique interval N-models in §4. We transform a general circular-arc graph G into a circle graph G_c and separate this circle graph into s-inseparable components. We then show that the corresponding induced subgraphs in G of these components in G_c have unique N-models. These N-models can be used to compose an N-model for the original circular-arc graph G.

4. A characterization of uniquely representable interval graphs. In this section, we present a proof that s-inseparable interval graphs have unique interval N-models. This analysis is of interest in its own right since it does not involve those powerful arguments borrowed from circle graphs [5]. The characterization of circular-arc graphs that have unique circular-arc N-models is given in §6.

Given an interval N-model R of an interval graph, the head of an interval is its left endpoint and the tail is its right endpoint. We shall not be concerned about the notion of "strongly adjacent" vertices for interval graphs since there is no corresponding geometric meaning on intervals.

THEOREM 4.1. *If an interval graph G does not contain any substitution, then it has a unique interval N-model.*

Proof. Associate with G the following graph G_c, whose vertex set is $V(G)$ and whose edge set consists of those strictly adjacent pairs of vertices in G. Since G does not contain any substitution, G must be connected. An interval is said to be *minimal* in G if it does not contain any other interval in G. An interval is said to be *simplicial* if it is minimal and is contained in all intervals that overlap with it. Let V_1 be the set of all simplicial intervals in G. Note that intervals in V_1 must be independent.

We first show that $G_c\backslash V_1$ is connected. Suppose otherwise. Let i be a minimal arc in $G\backslash V_1$. Let C_1 be a connected component in $G_c\backslash V_1$ containing i. Let j be any interval in $G\backslash(V_1 \cup C_1)$ that overlaps with some interval, say w, in C_1 (such a j must exist; otherwise, G is disconnected). Since j does not strictly overlap with w, we must have j contains w. Then all vertices of C_1 that overlap with w must also overlap with j and, hence, are contained in j. Through breadth-first search, we can argue that all vertices in C_1 must be contained in j. Let V_1' be the subset of simplicial intervals in V_1 that overlap with some vertices in C_1. Then, all intervals in V_1' are also contained in j. Now, if $|C_1| = |\{i\}| = 1$, then since i is not simplicial in G, there must exist an interval i' in V_1' contained in i. Therefore, $|C_1 \cup V_1'| \geq 2$ and $C_1 \cup V_1'$ is a nontrivial module in G, a contradiction.

We now show that there is a unique endpoint ordering for intervals in $G\backslash V_1$ in any N-model for G. This would imply that there is a unique N-model for G, since there is a unique way to insert intervals of V_1 once a model for $G\backslash V_1$ is formed. Let T be a spanning tree for the graph $G_c\backslash V_1$. Pick an arbitrary node of T as the root and perform a breadth-first search on T. Let $\pi = i_1, \ldots, i_k$ be the resulting vertex ordering of $G_c\backslash V_1$. Then π satisfies that each i_s strictly overlaps with some interval in $\{i_1, \ldots, i_{s-1}\}$. Since i_2 strictly overlaps with i_1, one may assume the endpoint order of i_1, i_2 is $h(i_2)h(i_1)t(i_2)t(i_1)$.

We now prove inductively that for each i_s, $s = 3, \ldots, k$, there is a unique way to insert the endpoints of i_s into the unique label sequence for $H_{s-1} = \{i_1, \ldots, i_{s-1}\}$. It suffices to show that, for any interval j in H_{s-1}, there is a unique left–right order between $\{h(i_s), t(i_s)\}$ and $\{h(j), t(j)\}$. If j contains i_s, then the order for i_s and j is $h(j)h(i_s)t(i_s)t(j)$; if j is contained in i_s, then the order is $h(i_s)h(j)t(j)t(i_s)$. Hence, consider the case that j strictly overlaps with i_s. Let j' be another interval of H_s that strictly overlaps with j. Without loss of generality, assume the endpoint order of j, j' is $h(j')h(j)t(j')t(j)$. If i_s does not overlap with j', then the order is $h(j)h(i_s)t(j)t(i_s)$. If i_s contains j' or is contained in j', the order is $h(i_s)h(j)t(i_s)t(j)$. Hence, assume i_s strictly overlaps with j'. If $N(i_s)\backslash[N(j) \cup N(j')] \neq \emptyset$, then the order is $h(j)h(i_s)t(j)t(i_s)$; otherwise, we have $N(j)\backslash[N(i_s) \cup N(j')] \neq \emptyset$, and the order is $h(i_s)h(j)t(i_s)t(j)$. □

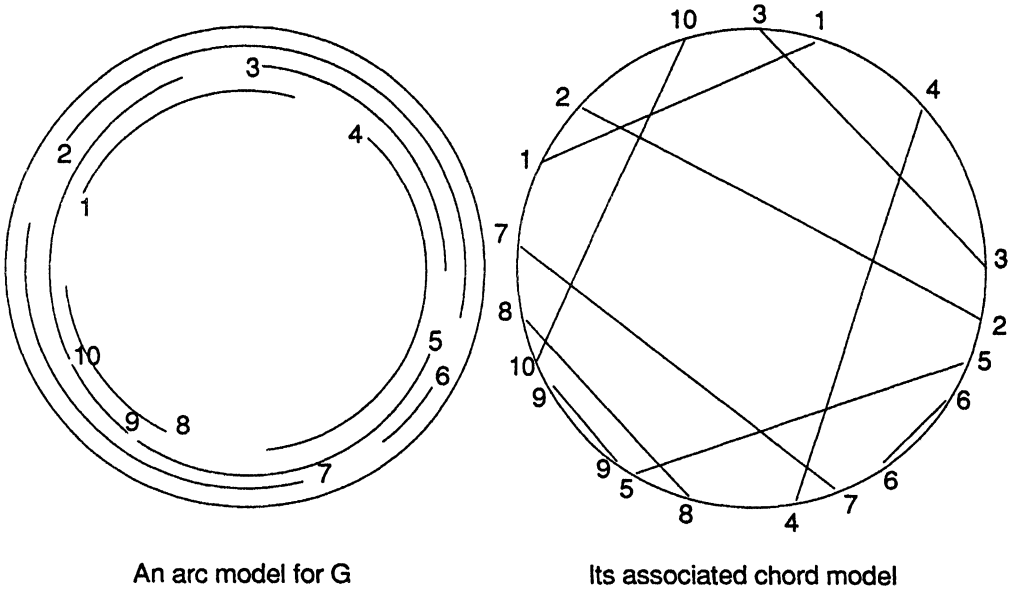


FIG. 5.1. *The chord model associated with a circular-arc model.*

5. The circle graph G_c associated with a circular-arc graph G. Graph decomposition has been successfully adopted to tackle the recognition problems for special classes of intersection graphs (see [7]). Gabor, Hsu, and Supowit [5] recently gave an $O(m \cdot n)$ decomposition algorithm for recognizing circle graphs. They made use of the following properties.

THEOREM 5.1 [5]. (1) *A graph G is a circle graph iff every j-inseparable component of G is a circle graph.*

(2) *Each j-inseparable circle graph has a unique chord model.*

We shall apply similar decomposition approaches to circular-arc graphs to obtain components that have unique N-models. In our approach, we first transform the proposed circular-arc graph to a circle graph and then make use of Theorem 5.1.

DEFINITION. *Associate with each graph G the graph G_c that has the same vertex set as G such that two vertices in G_c are adjacent iff they are strictly but not strongly adjacent in G.*

Consider a circular-arc graph G with an N-model R. Associate with each arc in R a chord that connects its two endpoints The resulting chord model D is called an *associated chord model* for G. Since two arcs cross in R exactly when their corresponding chords in D cross, this chord model gives rise to the circle graph G_c. An example is shown in Fig. 5.1. Although there can be many associated chord models for G, they all give rise to the unique circle graph G_c defined above.

Our decomposition approach is motivated by the analysis below. Given a circular-arc graph G and an N-model R for G, the following two types of structures in R can create multiple chord models by reversing the order of arc endpoints in a subset of $V(G)$. Let A be a subset of vertices in G. For convenience, we shall refer to "an arc in A" instead of "an arc in R corresponding to a vertex in A." (The notions "vertex," "arc," and "chord" will be used interchangeably whenever no confusion arises.)

Type I. There exist a segment (a, b) and a partition of $V(G)$ into two sets A and B such that all endpoints of arcs in A are contained in (a, b) and all endpoints of arcs in B are contained in (b, a) and $|A| \geq 2$, $|B| \geq 2$ (see Fig. 5.2).

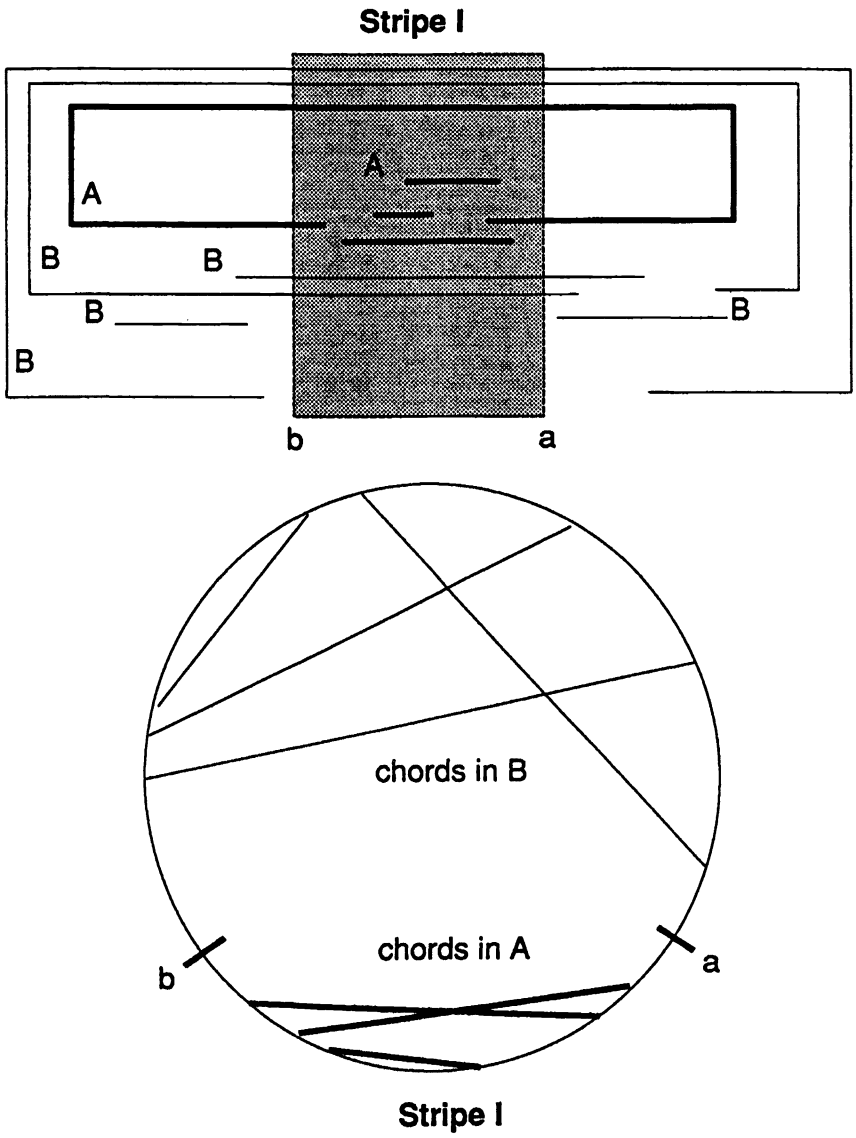


FIG. 5.2. *A circular-arc graph containing a type* I *structure.*

Type II. There exist two disjoint segments (a_1, b_1), (a_2, b_2) and a partition of $V(G)$ into A and B such that each arc in A has one endpoint in (a_1, b_1) and the other endpoint in (a_2, b_2) but none of the endpoints of arcs in B are in $(a_1, b_1) \cup (a_2, b_2)$ and $|A| \geq 2$, $|B| \geq 1$ (see Fig. 5.3).

Note that, in both cases, if G is an interval graph, then it must contain a substitution.

As one can see in Figs. 5.3 and 5.4, type I structures give rise to disconnected G_c and type II structures give rise to nontrivial modules in G_c. These two types of structures can be eliminated by applying the modular decomposition on the transformed graph G_c. The corresponding induced subgraph in G of the s-inseparable components in G_c can be shown to have unique N-models. The above discussion of type I and type II structures is there to give

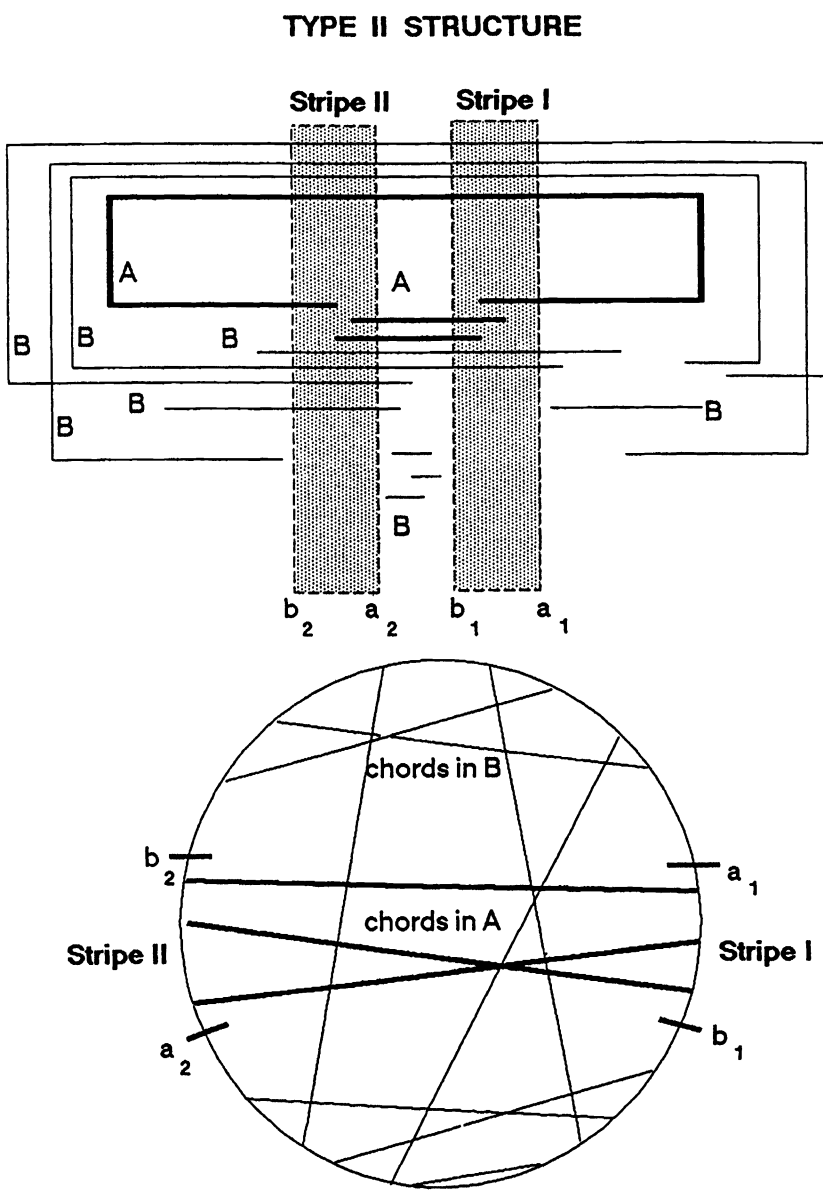


FIG. 5.3. *A circular-arc graph containing a type* II *structure.*

the reader guidance and is not necessary for the remainder of the proof. The following lemma states that, without loss of generality, we can assume the complement of G_c is connected.

LEMMA 5.2. *Let G be a graph with a disconnected $\overline{G}_c$. Let $C_1, C_2, \ldots, C_k$ be the components of $\overline{G}_c$. Let H_c^i and H^i, $i = 1, \ldots, k$, be their corresponding induced subgraphs in G_c and G, respectively. Then G is a circular-arc graph iff each of the H^i, $i = 1, \ldots, k$, is a circular-arc graph.*

Proof. The "only if" part is trivial. Consider the "if" part. Each H_c^i, $i = 1, \ldots, k$, is a permutation graph since there is a chord intersecting all of its chords in any model of G_c. Let the circular endpoint sequence H^i be $S^i(A)S^i(B)$, where every arc has exactly one endpoint

in $S^i(A)$ and the other one in $S^i(B)$. Then $S^1(A)S^2(A), \ldots, S^k(A)S^1(B)S^2(B), \ldots S^k(B)$ is a valid circular label sequence of an arc model for G. □

The transformation from an associated chord model of G_c back to an N-model for G involves the decision as to on which side of the chord the corresponding arc should be placed. This can be uniquely determined as shown in the following lemma.

LEMMA 5.3. *Let G be a circular-arc graph with a connected $\overline{G}_c$. Let D be an associated chord model of G_c. Then D determines a unique N-model R of G.*

Proof. Fix the placement of any arc u in the circle first. Let U_1 be the set of all vertices adjacent to u in $\overline{G}_c$. Then every arc in U_1 can now be placed on the side of their corresponding chord opposite u (if it is independent of u or covers the circle with u) or on the same side as u (if it contains u or is contained in u). Since the complement of G_c is connected, one can argue through a breadth-first search in $\overline{G}_c$ from u that every arc in G can be placed on a unique side of its corresponding chord in D. □

Conversely, however, not every chord model of G_c is an associated chord model for G since the model could violate other adjacency relationships in G that are missing in G_c.

5.1. The series and parallel relationships among nonadjacent vertices in G_c. The missing adjacency relationships in G caused by the transformation can be completely captured by the "series and parallel" relationships in G_c.

DEFINITION. *Three chords $d_1 = (a_1, b_1)$, $d_2 = (a_2, b_2)$, and $d_3 = (a_3, b_3)$ in a circle are said to be* in parallel with d_2 between d_1 and d_3 (*denoted by $d_1|d_2|d_3$) if the clockwise order of their endpoints is $a_1a_2a_3b_3b_2b_1$. They are said to be* in series (*denoted by $d_1 - d_2 - d_3$) if the clockwise order is $a_1b_1a_2b_2a_3b_3$.*

The following lemma is easy to justify.

LEMMA 5.4. *Let D be any chord model associated with an N-model for G. Let v_i, v_j, and v_k be three vertices in G and d_i, d_j, d_k their corresponding chords in D. Then these three chords satisfy $d_i|d_j|d_k$ iff one of the following conditions holds:*

(1) *v_k contains v_j; v_j contains v_i.*
(2) *v_k contains v_j; v_j and v_i cover the circle.*
(3) *v_k is contained in v_j; v_i contains v_j.*
(4) *v_k is contained in v_j; v_i is independent of v_j.*
(5) *v_k is independent of v_j; v_j contains v_i.*
(6) *v_k is independent of v_j; v_j and v_i cover the circle.*
(7) *v_k and v_j cover the circle; v_j is independent of v_i.*
(8) *v_k and v_j cover the circle; v_i contains v_j.*

DEFINITION. *Three pairwise nonadjacent vertices v_i, v_j, and v_k in G_c are said to be* in parallel with v_j between v_i and v_k (*denoted by $v_i|v_j|v_k$) if their relationships in G satisfy one of the eight (mutually exclusive) conditions in Lemma* 5.4. *Three vertices v_i, v_j, and v_k are said to be* in parallel *if they are in parallel with one of them "between" the other two. Three pairwise nonadjacent vertices v_i, v_j, and v_k of G_c that are not in parallel are said to be* in series (*denoted by $v_i - v_j - v_k$). A set of more than three vertices are said to be in parallel (resp., in series) if every three vertices in the set are in parallel (resp., in series).*

Note that when $v_i - v_j - v_k$, there is nothing special about j rather than i or k. Consider any vertex u of G_c. Let I_u be the set of vertices not adjacent to u in G_c. Partition I_u into two subsets L_u and R_u such that L_u consists of all vertices in I_u that are either contained in u or strongly adjacent to u in G and R_u consists of those in I_u that are either not adjacent to u or containing u in G. These two sets L_u and R_u are referred to as the "two sides" of u. Note that when $v_i - v_j - v_k$, then any two of them are on the same side of the third one. It is easy to see that if $v_1 \in L_u$ and $v_2 \in R_u$, then $v_1|u|v_2$. The converse is also true as shown in the following lemma.

LEMMA 5.5. *If $v_i|v_j|v_k$ in G_c then* (i) *v_i and v_k are on different sides of v_j*; (ii) *v_j and v_k are on the same side of v_i. If $v_i - v_j - v_k$, then v_i and v_k are on the same side of v_j.*

Proof. Without loss of generality, assume v_i is in L_{v_j}. Consider their relationships in G.

First, assume $v_i|v_j|v_k$. If v_i is contained in v_j in G, then $v_i|v_j|v_k$ implies that either v_k contains v_j (condition (1) of Lemma 5.4) or v_k is independent of v_j (condition (5) of Lemma 5.4). In either case, v_k must be in R_{v_j} and R_{v_i}. Hence, (i) holds. Since v_i is in R_{v_j}, (ii) also holds. If, on the other hand, v_i is strongly adjacent to v_j, then either v_k contains v_j (condition (2) of Lemma 5.4) or v_k is independent of v_j (condition (6) of Lemma 5.4) and we arrive at the same conclusion.

Now, assume $v_i - v_j - v_k$. This implies that none of the eight conditions in the definition of parallel vertices given in Lemma 5.4 apply to v_i, v_j, and v_k. If v_i is contained in v_j, then because conditions (1) and (5) do not hold, we have that v_k is either contained in v_j or strongly adjacent to v_j. Hence, $v_k \in L_{v_j}$ and v_i, v_k are on the same side of v_j. If v_i is strongly adjacent to v_j, then because conditions (2) and (6) do not hold, we also have that v_k is either contained in v_j or strongly adjacent to v_j and thus, derive the same conclusion. □

5.2. Conformal models. We shall now characterize those models of G_c that can be transformed into N-models for G.

DEFINITION. *A model D of G_c is said to be* conformal to G (*in short*, conformal), *if for every vertex u of G the chords associated with vertices in L_u are on one side of d_u (the chord for u) and those in R_u are on the other side of d_u.*

Equivalently, by Lemma 5.5, a model D is conformal if it satisfies (5.1) in G_c,

(5.1) for any three pairwise nonadjacent vertices v_i, v_j, and v_k,
$v_i|v_j|v_k$ iff their corresponding chords in D satisfy $D_i|d_j|d_k$.

DEFINITION. *An arc model $R(H)$ for an induced subgraph H of G is an* induced N-model *for H if the arc overlapping relationships in $R(H)$ conform to the vertex adjacency relationships in G (rather than those in H). A chord model $D(H_c)$ for an induced subgraph H_c of G_c is* conformal *if it satisfies* (5.1) *in G_c. $D(H_c)$ is said to be* conformal to a vertex *u not in H_c if there exists a placement for the chord of u into $D(H_c)$ such that the resulting chord model satisfies* (5.1).

THEOREM 5.6. *Let G be a circular-arc graph with a connected $\overline{G}_c$. Then a model for G_c is conformal iff it is a chord model associated with an N-model for G.*

Proof. The "if" part is trivial by the definition of parallel vertices. Hence, consider the "only if" part. Let D be a conformal chord model for G_c. Let u be any vertex of G_c. Fix the arc placement of u in the circle first. Then start placing the remaining arcs one by one according to the breadth-first search procedure described in the proof of Lemma 5.3. Similar to Lemma 5.3, the arc placement is uniquely determined.

We show that the resulting model R is an arc model for G (such an arc model is necessarily an N-model for G). If every new arc placed satisfies its adjacency relationships with all arcs previously placed, then we are done. Hence, assume there are violations of the adjacency relationships. Since two adjacent vertices of G_c must have their chords cross in model D, they cannot create any violation. Therefore, assume the first violation is found between two nonadjacent vertices v_i and v_j of G_c when we try to place v_j (whereas the arc for v_i has already been placed). We shall derive a contradiction. Let S be the set of arcs successfully placed so far (immediately before v_j). Let $v_i P v_j$ be a shortest path in $\overline{G}_c$ connecting v_i and v_j through vertices in S. The path $v_i P v_j$ can be viewed as a series of "forced" placements starting with some vertex (the one first encountered in a breadth-first search from u) in P and ending with v_i, v_j.

If $|P| \geq 2$, then the subgraph H_c of G_c is induced on vertices of the path is j-inseparable. By Theorem 5.1, there is a unique chord model for H_c. By Lemma 5.3, this results in a unique arc model for H_c, which must be the same as the one induced by D. But the violation on v_i and v_j implies that such a placement produces a violation. Hence, G is not a circular-arc graph.

If $P = \{v_k\}$, then v_i, v_j, and v_k are pairwise nonadjacent in G_c. Hence, either (i) v_i, v_j, and v_k are in parallel or (ii) they are in series. In either case, it is easy to verify that once the two pairs of arcs $\{v_i, v_k\}$ and $\{v_j, v_k\}$ are correctly placed, it forces the placement of $\{v_i, v_j\}$ to be correct. A similar argument can be used for case (ii).

Hence, such a violation does not exist and our placement strategy successfully produces an N-model for G. □

By Theorem 5.7, the problem of testing whether G is a circular-arc graph is equivalent to that of checking if G_c has a conformal model. In the next section, we first consider the special case that G_c is s-inseparable. The general case is discussed in §§6 and 7.

5.3. S-inseparable components of G_c.

THEOREM 5.7. *Let G be a circular-arc graph. If G_c is s-inseparable, then G has a unique N-model.*

Proof. We show that G_c has a unique conformal model. First, decompose G_c into j-inseparable component graphs. Each of these components has a unique chord model by Theorem 5.1, which must be conformal (otherwise, G is not a circular-arc graph). Next, we show by induction on $|V(G_c)|$ that there is a unique way to compose a conformal model for G_c from those individual conformal models of the j-inseparable components of G_c. This is trivially true for graphs with no more than four vertices. Suppose this holds for all graphs whose cardinality is less than $|V(G_c)|(\geq 5)$. If G_c is j-inseparable, we are done. Otherwise, there exists a partition of $V(G_c)$ into V_0, V_1, V_2, and V_3 with $|V_0 \cup V_1| \geq 2$ and $|V_2 \cup V_3| \geq 2$ such that every possible edge exists between V_1, V_2 and no edge exists between V_0, $V_2 \cup V_3$, or between $V_0 \cup V_1$, V_3 in G_c. Because G_c is s-inseparable, none of these four sets can be empty. If $|V_1| > 1$ and there exists a vertex s in V_1 that is adjacent to all other vertices in $V_1 \cup V_2$, then it is easy to verify that $G_c \backslash \{s\}$ is s-inseparable. We can proceed with the construction below to show that $G_c \backslash \{s\}$ has a unique conformal model, and, furthermore, there is a unique way to insert the chord of s into this model. The argument is symmetric if such a vertex s belongs to V_2. Therefore, assume no such vertex s exists. Then it is easy to verify that the corresponding two component graphs, H_1 and H_2 of G_c (the existence of a substitution in either H_1 or H_2 would imply the existence of one in G_c), are s-inseparable.

By induction, both H_1 and H_2 have unique conformal chord models D_1 and D_2, respectively. First, assume $|V_0 \cup V_1| \geq 3$ and $|V_2 \cup V_3| \geq 3$. Now, there must exist a chord d_0 from V_0 and a chord d_1 from V_1 that do not intersect each other (otherwise, either V_0 or V_1 has size ≥ 2 and creates a substitution). Similarly, there exist a chord d_2 from V_2 and a chord d_3 from V_3 that do not intersect each other. There are four possible placements in composing chords in D_1 and D_2 to form a model for G_c. Each of these placements creates different series or parallel relationships among $\{d_0, d_1, d_3\}$ and $\{d_0, d_2, d_3\}$. Hence, the vertex adjacency relationships of G dictates that there is a unique way to compose D_1 and D_2 (the fact that G is a circular-arc graph guarantees that there is at least one feasible placement from these two models).

Now consider the degenerate case where either $|V_0| = |V_1| = 1$ or $|V_2| = |V_3| = 1$. Suppose there are two noncrossing chords d_2 and d_3 in D_2. The two placements of d_0 create different $\{d_0, d_2, d_3\}$ relations, so they cannot both be N-models. If no such noncrossing pair exists, then either V_2 or V_3 is a module or (if both have just one vertex) G is a P_4. □

COROLLARY 5.8. *Let G be a circular-arc graph with a connected G_c. Then each induced subgraph H of G corresponding to the representative graph H_c of a parallel or neighborhood module M in G_c has a unique induced N-model.*

Proof. Note that since G_c is connected, each proper submodule of G_c must correspond to a permutation graph. If M is a neighborhood module, then H_c is s-inseparable. Hence, the corollary is true by Theorem 5.8. If M is a parallel module (M cannot be $V(G_c)$), then H_c consists of a set of parallel vertices. Since the chords in any conformal model for H_c must follow the unique linear order dictated by the parallel relationships of vertices in H, there is a unique conformal model $D(H)$ for H_c. Since the complement of H_c is connected, $D(H)$ gives rise to a unique N-model for H by Lemma 5.3. □

This corollary will be used at the end of §6 to compose a conformal model for G_c. The general scheme of our transformed decomposition can be summarized as follows. Take a proposed circular-arc graph G, transform it into the graph G_c, and perform a decomposition on G_c to obtain components that correspond to uniquely representable induced subgraphs for G. If G is indeed a circular-arc graph, then we find a conformal model of G_c and transform it back to an arc model for G. Otherwise we could either detect a contradiction at an intermediate step or, at the end, construct an arc model for G that violates its vertex adjacency relationships. The transformation enables us to use well-known techniques for decomposing circle graphs.

We shall first illustrate our decomposition approach in §6 on those circular-arc graphs whose associated circle graphs are connected and then describe the reduction from general circular-arc graphs to the above subclass in §7.

6. Consistent decomposition of a connected G_c. Throughout this section we assume the graph G satisfies the following:

(6.1) Both $\overline{G}_c$ and G_c are connected.

The general case is discussed in §7. We shall describe a decomposition scheme on G_c that serves the following purposes: (a) yields components that are s-inseparable, (b) provides enough information on constructing a conformal model for G_c when G is a circular-arc graph, and (c) gives rise to a unique decomposition tree.

The example in Fig. 6.1 points out a major drawback of the ordinary modular decomposition when applied to G_c. Although the set $\{1, 2, 3\}$ is a maximal submodule in the neighborhood module $V(G_c)$, their placements are quite different due to the additional "conformity" constraint. A closer look at the type II structures reveals that they give rise to modules of a very special kind—those modules whose endpoint labels consists of two consecutive subsequences in any N-model (such modules will be called consistent modules in the next section). Therefore, we need to devise a decomposition scheme that accommodates this requirement.

6.1. Consistent modules of G_c. Let M be any subset of $V(G_c)$. Let $I(M)$ denote the set of vertices in $V(G_c)\backslash M$ that are not adjacent to all vertices of M in G_c. Note that M is a module iff no vertex in $I(M)$ is adjacent to any vertex of M.

DEFINITION. *A subset M of $V(G_c)$ is* consistent with *a subset $M' \subseteq I(M)$ if the following three conditions hold*:

(6.2) All vertices of M are on the same side of every vertex v in M' (in particular, M is a module).

(6.3) If $v_1 - u - v_2$ (resp., $v_1|u|v_2$ or $u|v_1|v_2$) for some u in M and v_1, v_2 in M', then $v_1 - u_1 - v_2$ (resp., $v_1|u_1|v_2$ or $u_1|v_1|v_2$) for every u_1 in M.

(6.4) There exist no u_1, u_2 in M and v in M' such that $v - u_1 - u_2$.

M is consistent *in G_c if it is consistent with $I(M)$.*

Note that any module (other than the whole graph) in a connected circle graph must be a permutation graph. Since it is possible to draw a chord that crosses all chords in a permutation model, a permutation graph does not contain three vertices in series.

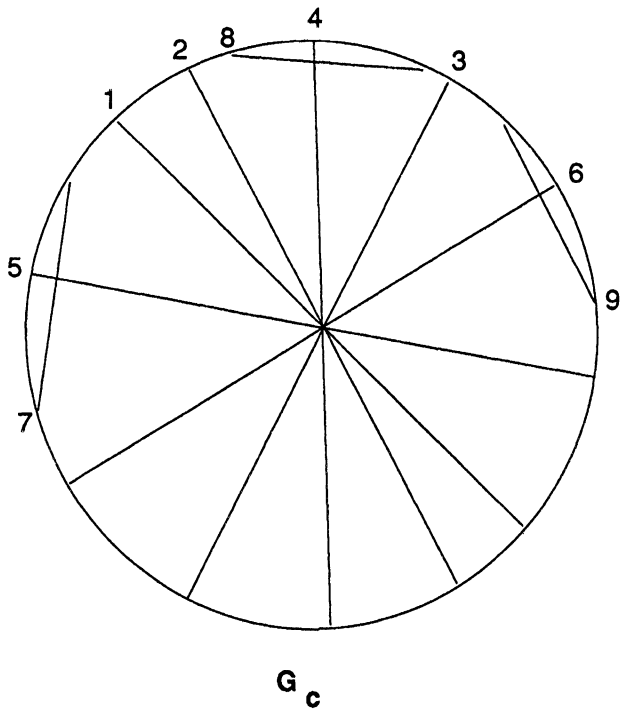


FIG. 6.1. *A maximal submodule that is not consistent. The partition of $V(G_c)$ into maximal submodules is* {1, 2, 3}, {4}, {5}, {6}, {7}, {8}, {9}. *The partition of $V(G_c)$ into maximal consistent submodules is* {1, 2}, {3}, {4}, {5}, {6}, {7}, {8}, {9}. *Hence the submodule* {1, 2, 3} *is not consistent.*

LEMMA 6.1. *Let M be a consistent submodule of $V(G_c)$ that gives rise to a permutation graph, then all submodules resulting from the ordinary substitution decomposition of M are consistent.*

Proof. Let $M_1, \ldots, M_k$ be the children submodules of M. Then each M_i is consistent with $I(M)$. We shall show that each M_i is consistent with $I(M_i)$. Since M gives rise to a permutation graph, there do not exist three vertices in series in M. Hence (6.4) holds for each M_i. To show that (6.2), (6.3) hold for M_i, we consider the following three cases:

(1) M is a parallel module. Then each M_i is a connected component of M. The parallel relationships dictate that there is a unique linear order for vertices from different M_i's in any conformal model. Hence, (6.2) and (6.3) hold.

(2) M is a series module. Then each M_i is a connected component in the complement of $G_c[M]$ and $I(M_i) = I(M)$. Since M is consistent, each M_i is consistent with $I(M)$, the same as $I(M_i)$.

(3) M is a neighborhood module. Then $M_1, \ldots, M_k$ are the maximal submodules of M. Suppose one of them, say M_i, is not consistent.

If M_i violates (6.2) then there exist a vertex v in $I(M_i)\backslash I(M)$ and two vertices u, u' in M_i such that $u|v|u'$. Let U be the set of all vertices in $I(M_i)$ between u and u' (and in parallel with u, u'). Since M is consistent, $U \subseteq M$. Let t be any vertex in $M\backslash M_i$ adjacent to u in G_c. Because M_i is a module, t is also adjacent to u'. By considering any chord model D for G_c, it is easy to argue that t must be adjacent to every vertex in U. Now, let v' be any vertex in $I(M_i)\backslash U$. Then, u and u' must be on the same side of v' and we have either $v'|u|u'$ or $u|u'|v'$. In any case v' cannot be adjacent to any vertex in U. Furthermore,

$M\backslash(M_i \cup U) \neq \emptyset$; otherwise, M is not connected. Hence, we conclude that $M_i \cup U$ is a module in M, contradictory to the fact that M_i is a maximal submodule of M.

If M_i violates (6.3), then a similar contradiction can be obtained. □

COROLLARY 6.2. *If G_c is a permutation graph, then all submodules in the substitution decomposition tree are consistent (note that this holds even if G_c is disconnected).*

6.2. Consistent partition of the neighborhood module $V(G_c)$. By assuming (6.1), $V(G_c)$ must be a neighborhood module. Let $M_1, \ldots, M_k$ be the maximal submodules that partition $V(G_c)$ in the "ordinary" neighborhood decomposition that are not necessarily consistent. Then each of them gives rise to a permutation graph. Note that k must be at least 3. The following lemma shows that it suffices for us to refine the series modules of $V(G_c)$.

LEMMA 6.3. *If a maximal submodule M_i of $V(G_c)$ is not consistent, then it is a series module.*

Proof. Consider any chord model D of G_c. Since M_i gives rise to a permutation graph, chords in M_i have one endpoint in a segment s_1 and the other endpoint in a disjoint segment s_2 of the circle. If those endpoints of M_i in s_1 as well as those in s_2 are consecutive in D, then one can easily verify that M_i is consistent.

Hence, without loss of generality, assume there is a chord $t = [a, b]$ in $V(G_c)\backslash M_i$ such that endpoint a is between two endpoints i, i' of M_i in segment s_1. If t crosses all chords of M_i, then endpoint b must fall in segment s_2. From model D, one can see that M_i is a series module. Now consider the case that $t \in I(M_i)$. Let $P = t \ldots t'u$ be a shortest path from t to M_i with $t' \notin M_i$, $u \in M_i$. Then one endpoint of t' falls between i, i' in segment s_1 and t' crosses all chords of M_i. Hence, the other endpoint of t' is in segment s_2. Since all chords of M_i crosses t', M_i must be a series module. □

THEOREM 6.4. *Let M_i be a maximal series submodule of the neighborhood module $V(G_c)$. Then there is a unique partition of M_i into maximal subsets $M^1, M^2, \ldots, M^r$, each of which is consistent in G_c.*

Proof. Consider the following relation "$\approx$" on M_i: $u \approx v$ iff (a) u, v are on the same side of every t in $I(M_i)$; (b) for any two t_1, t_2 in $I(M_i)$, $t_1|u|t_2$ iff $t_1|v|t_2$; $t_1 - u - t_2$ iff $t_1 - v - t_2$; and (c) there is no t in $I(M_i)$ such that $t - u - v$. It is not difficult to show that this is an equivalence relation. Let $M^1, \ldots, M^r$ be the unique partition of M_i into equivalence classes. By the definition of "$\approx$," these M^j's must be maximal subsets of M_i consistent with $I(M_i)$. Since M_i is a series module, $I(M^j) = I(M_i)$ for $j = 1, \ldots, r$. Hence, each M^j is consistent with $I(M^j)$. □

DEFINITION. *The* consistent partition *of $V(G_c)$ is the unique refinement of maximal submodules of $V(G_c)$ into their maximal consistent submodules as described in Theorem* 6.8. *A maximal submodule in the neighborhood partition of $V(G_c)$ is said to be* split *if it is refined in the consistent partition (an example is shown in Fig.* 6.1).

DEFINITION. *The* consistent decomposition tree *T has $V(G_c)$ as its root module, the maximal consistent submodules of $V(G_c)$ as the children submodules of $V(G_c)$, and the substitution decomposition tree of each children submodule M_i as the subtree of T with root module M_i. Every submodule associated with an internal node of the decomposition tree T is referred to as a* T-submodule.

Because the decomposition at each node is unique, the tree T is unique up to isomorphism. Since all children T-submodules of $V(G_c)$ are consistent and give rise to permutation graphs, by Lemma 6.5, all T-submodules of the tree T are consistent. We shall show that the representative graph of $V(G_c)$ has a unique conformal model. An important property of T-submodules is the following.

LEMMA 6.5. *Let M be a T-submodule that gives rise to a permutation graph. Let D be any conformal model for G_c. Then the endpoints of chords of M can be partitioned into two*

sets A_1, A_2 such that each chord of M has one endpoint in A_1 and the other in A_2 and those endpoints in A_1 (resp., A_2) are consecutive in D.

Proof. We first show that the lemma holds for each children T-submodule M^j of $V(G)$.

CLAIM. *Let $v = [w, x]$ be a chord in $V(G)\backslash M^j$ crossing all chords in M^j. Let (a_1, b_1) (resp., (a_2, b_2)) be the shortest segment containing w (resp., x) with endpoints from chords in M^j. Then, there must exist a chord in $I(M^j)$ whose two endpoints are contained either in (a_1, b_1) or in (a_2, b_2).*

Proof of Claim. Suppose otherwise. Let U_1 (resp., U_2) be the set of chords in D such that at least one of their endpoints falls in (a_1, b_1) (resp., (a_2, b_2)). Since we assume no chord of U_1 (resp., U_2) have both endpoints in (a_1, b_1) (resp., (a_2, b_2)), each chord of U must cross all chords in M^j. Therefore, all of their other endpoints must fall in (a_2, b_2) (resp., (a_1, b_1)). Hence, $U_1 = U_2$. From model D, it is easy to check that $M^j \cup U_1$ is consistent. Since $V(G_c)\backslash(M^j \cup U_1) \supseteq I(M^j) \neq \emptyset$, we have that $M^j \cup U_1$ is a nontrivial consistent submodule of $V(G_c)$, a contradiction. □

Let v' be a chord in $I(M)$ with its endpoints either in (a_1, b_1) or in (a_2, b_2). We shall find another chord t' in $I(M)$ whose endpoints are either in (b_1, a_2) or in (b_2, a_1). If none of the segments (b_1, a_2) and (b_2, a_1) contain endpoints of chords not in M^j, the lemma holds. Hence, assume to the contrary, there is a chord $t = (p, q)$ in $V(G)\backslash M^j$ such that one of its endpoints, say p, falls in (b_2, a_1). Let (c_1, d_1) be the shortest segment containing p with endpoints in M^j. If $q \in (c_1, d_1)$, let $t' = t$. Otherwise, t crosses M^j and q must fall in (b_1, a_2). Let (c_2, d_2) be the shortest segment containing q with endpoints in M^j (see Fig. 6.2). By the claim, the existence of t implies that there exists a chord t' in $I(M)$ whose endpoints are contained in either (c_1, d_1) or (c_2, d_2). In either case, one can easily verify that t' and v' are in series with some chord in M^j, and they are in parallel with some other chord in M^j, contradictory to the consistency of M^j.

Next, we show that Lemma 6.5 holds for each T-submodule of M^j. By induction, we need only show this for every children T-submodule of M^j. By Lemma 6.1, these T-submodules are all consistent. If M^j is a parallel module, then the lemma is implied by the unique parallel linear order of the T-submodules of M^j. If M^j is a neighborhood module, then the above argument on M^j itself can be applied here. Hence, consider the case that M^j is a series module. If any child T-submodule M' of M^j does not satisfy the lemma, then M' must be a series module. But then, M' should not have been a component in the series decomposition of M^j (each component of a series module must have a connected complement). □

THEOREM 6.6. *Consider a neighborhood module $V(G_c)$. Let $V' = \{v_1, \ldots, v_r\}$ be a set of representatives from the maximal consistent submodules $M^1, \ldots, M^p$ in the consistent partition of $V(G_c)$. Then there is a unique conformal model for V'. Furthermore, this model is independent of the v_i's selected (we shall refer to the subgraph induced on V' as the* representative graph for $V(G_c)$).

Proof. By Lemma 6.5, the consecutive endpoint property of the M^j's implies that such a model is independent of the v_i's selected. The endpoints of any subset V_1 of V' divide the circle into a collection of segments. Chords not in V_1 can be inserted into a chord model for V_1 by specifying the pair of segments that contain its two endpoints.

CLAIM 1. *There do not exist four chords u_1, u_2, u_3, and u_4 with endpoints a_i, b_i, $i = 1, 2, 3, 4$ arranged as $a_1a_3a_2a_4b_1b_3b_2b_4$ in D such that u_1, u_2 belong to a maximal submodule M_i; u_3, u_4 belong to another maximal submodule M_j.*

Proof of Claim 1. If these chords exist, then any chord in $V(G_c)\backslash(M_i \cup M_j)$ that crosses M_i must also cross M_j and vice versa. This would make $M_i \cup M_j$ a module of $V(G_c)$ (note that the number of maximal consistent submodules of $V(G_c)$ is at least three). □

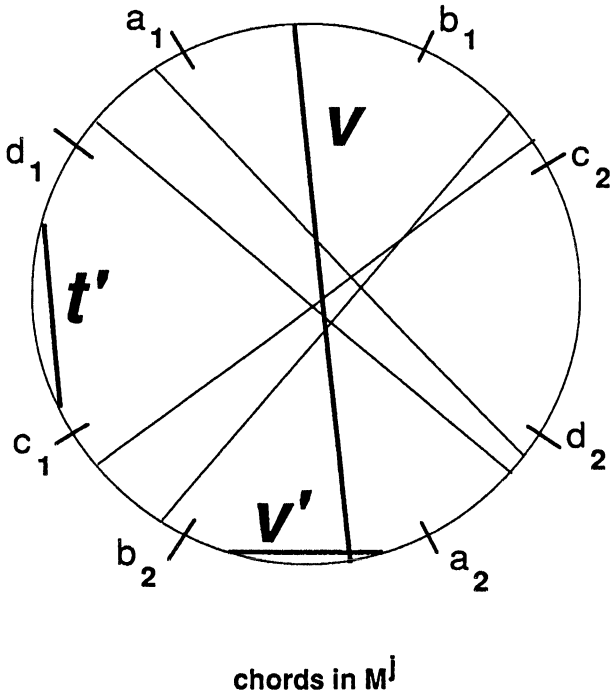


FIG. 6.2. *The four segments and* M^j.

Let M_i be any maximal submodule in the neighborhood partition (as opposed to the consistent partition) of $V(G_c)$. Let $v_{i_1}, \ldots, v_{i_r}$ be the representatives in V' from the maximal consistent submodules $M^{i_1}, \ldots, M^{i_r}$ of M_i.

CLAIM 2. *For any subset* V'' *of* V' *that constitutes a set of representatives for all maximal submodules except* M_i *in the neighborhood partition of* $V(G_c)$, *there is a unique conformal model for* $\{v_{i_1}, \ldots, v_{i_k}\} \cup V''$.

Proof of Claim 2. By Theorem 5.7, there is a unique conformal model for each $\{v_{i_m}\} \cup V''$, $m = 1, \ldots, k$. Since there is a conformal model for $\{v_{i_1}, \ldots, v_{i_k}\} \cup V''$, the endpoint order for chords in V'' in each model $\{v_{i_m}\} \cup V''$, $m = 1, \ldots, k$ must be the same. The endpoints of chords in V'' divide the circle into a collection C of segments. Therefore, there is a unique pair of segments in C for the chord of each v_{i_m}, $m = 1, \ldots, k$. It remains to show that no two representative chords form M_i can have one of their endpoints sharing the same segment in C (since M_i is a module, these two chords would have to share the same pair of segments in C).

Suppose there are two chords, say v_{i_1}, v_{i_2}, sharing the same pair of segments of C. By Lemma 6.3, M_i is a series module. Hence, v_{i_1} must cross v_{i_2}. If there exists a chord v in $I(M_i)$ such that one of its endpoints separates those of v_{i_1} and v_{i_2} in one of the segment pair, say s, then following the proof of Lemma 6.3, one can argue that there exist a chord v' in $V(G)\backslash M$ with an endpoint in s that crosses both v_{i_1} and v_{i_2}. Let v_s be the representative in V'' of the module containing v' (since the endpoints of v_s are used in defining the segments in C, v_s cannot be the same as v'). Then v_{i_1}, v_{i_2}, v', and v_s form a forbidden configuration described in Claim 1. □

By Claim 2, there will be no ambiguity in placing the representative chords $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ of any maximal submodule M into the unique model for any subset V'' of V' described in the beginning of the proof. Hence, we can construct a unique model for V' as follows. Start by choosing an arbitrary V'' and construct the unique conformal model for V''. Let $M_1, \ldots, M_s$

be the maximal submodules of $V(G_c)$ in the neighborhood partition. Place the chords in $V' \cap M_i$, $i = 1, \ldots, s$ into the current conformal model one by one. This concludes the proof of Theorem 6.6. □

Suppose G_c has a conformal model. To ensure that a model $D(M)$ for a T-submodule M is consistent with respect to chords not in M, it may appear that we need to check the consistency of $D(M)$ with respect to each vertex in $I(M)$. Since each T-submodule M other than $V(G_c)$ is consistent and gives rise to a permutation graph, it can be easily shown that $D(M)$ is consistent in G_c iff $D(M)$ is consistent with respect to an arbitrary vertex in $I(M)$. Hence, we can construct a conformal model for G_c as follows.

For each T-submodule M, let $V(M)$ be a set of representatives in a representative graph for M. Choose a vertex $u(M)$ in $I(M)$ and construct a conformal model for the subgraph induced on $V(M) \cup \{u(M)\}$. By Lemma 6.5, a model for G_c can be composed in a bottom-up fashion along the tree T by recursively replacing the representative chord with its corresponding permutation chord model as follows: (a) when M is a parallel module, we place the chord models of its children T-submodules according to the unique linear order dictated by the parallel relationships; (b) when M is a neighborhood module, we place them according to the unique N-model described in Corollary 5.8; and (c) when M is a series module, its children chord models are made to intersect each other's.

THEOREM 6.7. *Let G be a graph satisfying* (6.1). *Let T be the unique decomposition tree for G_c. Then G is a circular-arc graph iff for every T-submodule M there is a conformal model for the subgraph induced on $V(M) \cup \{u(M)\}$.*

7. Decomposition of a circular-arc graph G with a disconnected G_c. In §6 we assume G_c is connected and apply the consistent decomposition to eliminate type II structures in G. In case G_c is not connected (type I structure exists), one needs to arrange the relative positions of its components according to the series and parallel relationships in G, which makes the model construction for each component of G_c more involved. We shall adopt the following approach:

(a) Represent the series and parallel relationships among the components by a tree.

(b) Construct a unique conformal model for each component node of the tree in (1).

(c) Combine the individual component models together based on the tree in (1) to form a conformal model for G_c.

Figure 7.1 illustrates the importance of keeping track of such relationships by showing an example of a noncircular-arc graph G with disconnected G_c. In this example, we have two conflicting series relationships 6-2-4 and 6-3-5.

7.1. The unique tree T_{NS} representing the series and parallel relationships among components in G_c. Consider a circular-arc graph G with a disconnected G_c. Let $C_1, \ldots, C_k$ be the connected components of G_c.

DEFINITION. *Three components C_i, C_j, and C_k are said to be* in parallel (*denoted by $C_i|C_j|C_k$*) *if there exist three vertices v_i, v_j, and v_k from C_i, C_j, and C_k, respectively, satisfying $v_i|v_j|v_k$. Three components that are not in parallel are said to be* in series (*denoted by $C_i - C_j - C_k$*) (*if $C_i - C_j - C_k$, then any three vertices v_i, v_j, and v_k from C_i, C_j, and C_k, respectively, must also be in series*). *A collection of components are in series if every three of them are.*

DEFINITION. *A component C_i is said to be* near *another component C_j if there exists no component C_k such that $C_i|C_k|C_j$. Define a collection Q of components to be an NS-*collection *if the components in Q are in series and every two components are near each other.*

Let $N_1, N_2, \ldots, N_r$ be the maximal NS-collections of G_c. It is easy to check that the number of maximal NS-collection of G_c is no greater than the number of connected components in G_c. An example is given in Fig. 7.2.

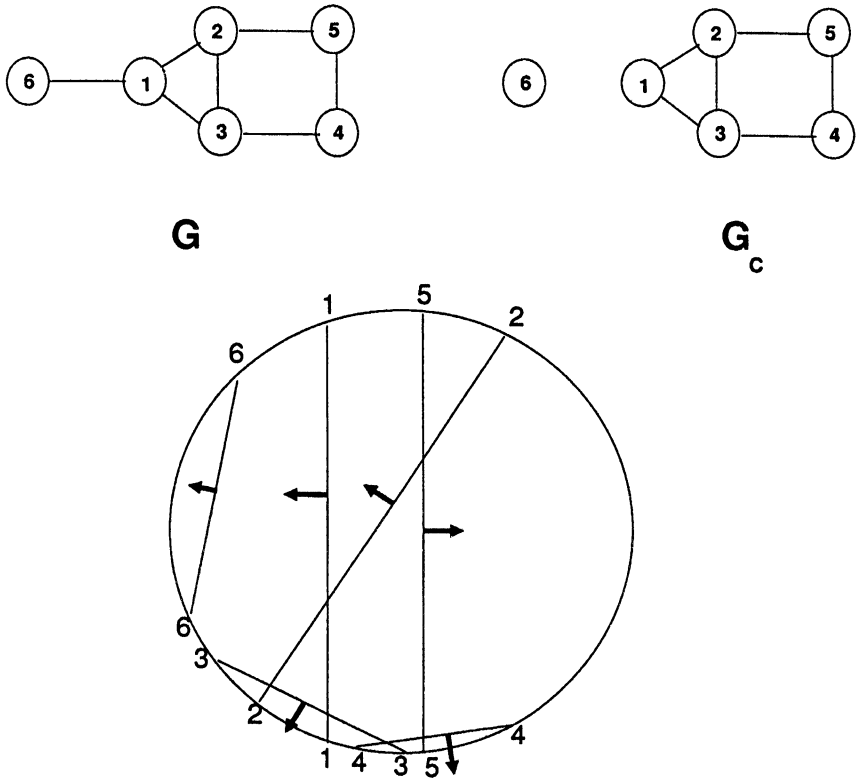


FIG. 7.1. *A noncircular-arc graph G with a disconnected G_c. The five vertices component of G_c gives a uniquely representable circular-arc graph. Because* 6 *and* 1 *are adjacent in G. we have to place arc* 6 *to the left of its chord. But then* 6 *and* 2 *have to be adjacent, a contradiction.*

The following relationships on the components can be easily justified.

LEMMA 7.1. *Let C_i be any component of G_c. Let N_j, N_j' be two maximal NS-collection containing C_i. Then, for any two components C, C′ of $N_j \backslash \{C_i\}$ and $N_j' \backslash \{C_i\}$, respectively, we have $C|C_i|C'$.*

Now, construct a bipartite graph T_{NS} with vertex set on the components C_i's of G_c and the maximal NS-collections N_i's by connecting each N_i to every component in N_i by an edge. An example is given in Fig. 7.3.

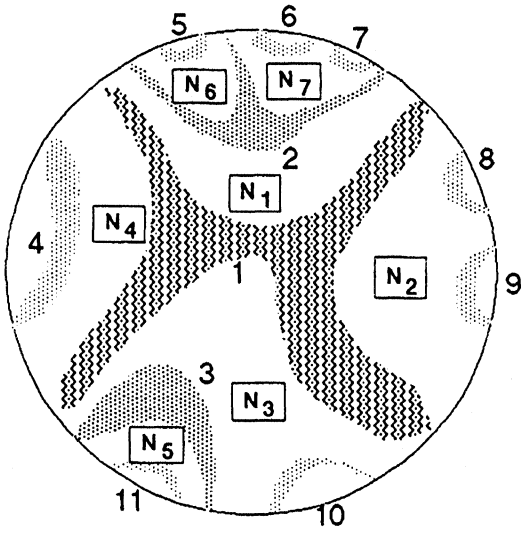
THEOREM 7.2. *The graph T_{NS} is a tree (called the* series-parallel tree, *or SP-tree, of G_c).*

Proof. Suppose there exists a cycle $C_1, N_1, C_2, N_2, \ldots, C_m, N_m, C_1$. By Lemma 7.1, we have (a) $C_1|C_2|C_3, C_2|C_3|C_4, \ldots, C_{m-1}|C_m|C_1$ and (b) $C_m|C_1|C_2$. However, (a) implies that C_2 and C_m are on the same side of C_1, which contradicts (b). □

7.2. C-inseparable graphs. In this section, we show that the problem of constructing a conformal model for G_c can be reduced to that of constructing a conformal model for each of its c-inseparable components defined below.

DEFINITION. *For each component C_i, define a* c-inseparable component *G_i of G_c to be the union of C_i together with any vertex from N_j for each maximal NS-collection N_j containing C_i (as shown in Fig.* 7.4). *G_c is c-inseparable if there exists a component C_i such that $G_c = G_i$.*

As far as the conformal model S of G_i is concerned, any vertex from any component of N_j will result in the same model. Therefore, the selection is immaterial. If G_c is connected, then it



G_c

$N_1 = \{ C_1, C_2 \}$

$N_2 = \{ C_1, C_8, C_9 \}$

$N_3 = \{ C_1, C_3, C_{10} \}$

$N_4 = \{ C_1, C_4 \}$

$N_5 = \{ C_3, C_{11} \}$

$N_6 = \{ C_2, C_5 \}$

$N_7 = \{ C_2, C_6, C_7 \}$

FIG. 7.2. *The maximal NS-collection of* G_c.

is clearly c-inseparable by this definition. The following two lemmas provide characterizations of c-inseparable graphs. The first one can be easily justified.

LEMMA 7.3. *The* T_{NS} *tree for a c-inseparable graph* $G_c (= G_i)$ *satisfies that*

(1) *every component of* G_c, *except possibly* C_i, *is a leaf in* T_{NS} *containing exactly one vertex of* G_c.

(2) *every maximal NS-collection of* G_c *consists of* C_i *and a leaf component.*

THEOREM 7.4. *Consider a circular-arc graph G. Suppose* G_c *is c-inseparable and disconnected. Let* C_i *be the component such that* $G_i = G_c$. *Let K be the set of vertices of* G_c *not in* C_i. *Then any conformal model for* G_c *satisfies that*

(1) *between any two chords u, v in K there is a chord d in* C_i *such that* $u|d|v$.

(2) *those chords in K are in series following a unique (up to rotation and reversal) circular order around the circle.*

Proof. Each vertex in K forms a single vertex component of G_c. Let u, v be two vertices in K. By Lemmas 7.1 and 7.3, $u|C_i|v$. Hence, condition (1) holds.

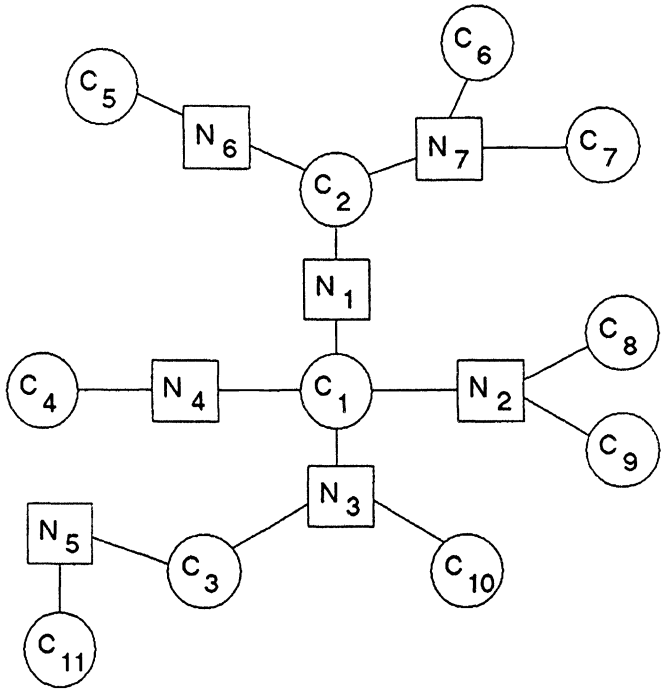


FIG. 7.3. *The series parallel tree T_{NS} of a graph G_c.*

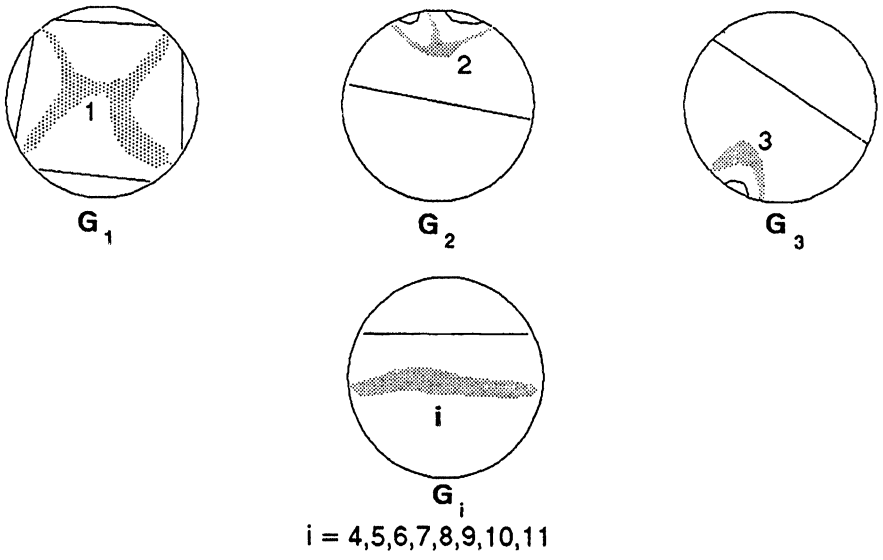


FIG. 7.4. *The c-inseparable components of a graph G_c.*

Condition (2) can be proved by the following set partitioning argument. Each vertex u in C_i partitions K into two sets, L_u and R_u. By considering vertices of C_i one by one and refining the partition on K, we can eventually conclude that, in the final partition of K, every set is a singleton by (1). □

THEOREM 7.5. *Assume $\overline{G}_c$ is connected. Then, G_c has a conformal model iff each of its c-inseparable component G_i has a conformal model.*

Proof. The "only if" part is trivial since each G_i is an induced subgraph of G_c. Hence, consider the "if" part.

We prove this by induction on $|V(G_c)|$. Assume G_c is not c-inseparable; otherwise, the theorem is trivially true. By Lemma 7.3, we have the following cases.

Case 1. T_{NS} contains a leaf component C whose size is greater than 1. Let u_1 be any vertex in C and u_2 be any vertex not in C. Let H_1 be the subgraph of G_c induced on $[V(G_c)\backslash C]\cup\{u_1\}$ and H_2 be the subgraph of G_c induced on $C\cup\{u_2\}$.

Case 2. T_{NS} contains a maximal NS-collection N that contains more than two components and at least one of them, say C, is not a leaf. Now, deleting the edge between N and C will result in two trees T_1 (the one containing N) and T_2 (the one containing C). Let u_1 be any vertex in C and u_2 be any vertex contained in a component in T_1. Let H_1 be the subgraph of G_c induced on the union of $\{u_1\}$ and those components in T_1. Let H_2 be the subgraph of G_c induced on the union of $\{u_2\}$ and those components in T_2.

By Lemma 7.1, all vertices of $V(H_j)\backslash\{u_j\}$ are on the same side of u_j in G_c for $j=1,2$. By induction, each of H_1 and H_2 has a conformal model D_1, D_2 with endpoint sequences, S_1ab and S_1cd, respectively, where $[a,b]$ is the chord for u_1, $[c,d]$ is the chord for u_2. The reason that a, b are consecutive in D_1 is because all other chords are on the same side of $[a,b]$ by the conformity of D_1. The same argument goes for c, d.

Let D be the chord model with endpoint sequence S_1S_2. D satisfies that any two chords of A are on the same side of every chord in B and vice versa. It also satisfies the intersection relationships of G_c. Hence, it is easy to check that D satisfies (5.1). □

7.3. Constructing a conformal model for a c-inseparable G_c. In this section, assume G_c has a conformal model and is c-inseparable. Let C be the component of G_c of size greater than 1 (if no such component exists, then $|V(G_c)|\leq 3$ and the problem is trivial). Let K be the set of remaining vertices $u_1,\ldots,u_k$, each of which constitutes a single-vertex component of G_c and is indexed according to the unique circular order prescribed in Theorem 7.4. We shall first construct a consistent decomposition tree T for the subgraph $G_c[C]$ and then insert the vertices of K into conformal models of appropriate T-submodules. For any T-submodule M in C, let $I(M)$ be the set of vertices of C not adjacent to M. A module M is said to be *split* by a vertex $u\notin M$ if there exist vertices v, v' in M such that either (a) $u-v-v'$ or (b) there exists a $u'\notin M$ such that $u|v|u'$ and $u-v'-u'$.

LEMMA 7.6. *Let u be a vertex in K. Let M be a T-submodule and $M_1,\ldots,M_r$ be the children T-submodules of M. If M is split by u, then at most one of the M_i's is split by u.*

Proof. Suppose M_1 and M_2 are both split by u. Let D be a conformal model for G_c. Suppressing the endpoints on u, the endpoints of M_1 (resp., M_2) should be divided into two consecutive sequences A_1 and B_1 (resp., A_2 and B_2) because M_1 and M_2 are consistent in C (Lemma 6.5). Since all chords of C are on the same side of u, $h(u)$ and $t(u)$ should also be consecutive. The chord of u can be split M_1 only if either A_1 or B_1 is separated by $h(u), t(u)$. Since $h(u), t(u)$ can only separate one of the four sequences A_1, B_1, A_2, and B_2, at most one of M_1 and M_2 can be split by u. □

Following this lemma, we can find a unique path P from the root of T to a node representing a module, denoted by $M(u)$, which is split by u, but none of the children of $M(u)$ is split. If $M(u)$ is a series module, let $u(M)$ be a vertex outside $M\cup\{u\}$. Then divide those representative chords of M uniquely into two sets: those that are in between u, $u(M)$ and those that are in series with u and $u(M)$. Hence, consider the case that $M(u)$ is a parallel or a neighborhood module. Let D^* be the unique conformal models for the representative graph of $M(u)$. Then the series and parallel relationships of G dictate that there is a unique way to insert the chord of u into D^*. Therefore, we can insert each u of K uniquely into its corresponding T-submodule $M(u)$ in T. We call these adjusted modules *augmented T-submodules* (note that several u's

could be inserted into the same T-submodule, but no two of them can share the same segment). The resulting tree T' with vertices of K added is called the *consistent decomposition tree* of G_c.

8. Complexity analysis of the recognition algorithm. Our algorithm consists of the following five major steps. Each step can be shown to take at most $O(m \cdot n)$ time. Steps (4) and (5) are used for disconnected G_c. Described below is not the flow chart, but the main steps of the algorithm. After a model of G_c is constructed, we need to test if its corresponding arc model faithfully represents G.

(1) Transform G to G_c, determine L_v and R_v for each vertex v in G_c.

(2) Given a graph G satisfying (6.1), determine its consistent decomposition tree T.

(3) Determine a unique conformal chord model for the representative graph of each T-submodule by applying the join decomposition. Compose a conformal model for G_c in (4) from those of its T-submodules or declare that such a model does not exist.

(4) Given a graph G with a disconnected G_c, determine its series parallel tree T_{NS}.

(5) For each inseparable component G_i of G_c such that $G_c[C_i] = G_i$, construct the consistent decomposition tree T_i for $G_c[C_i]$. Determine the unique path for each vertex of K along T_i and construct the corresponding augmented T-submodule. Construct a conformal model for G_i. Finally, compose a model for G_c from those of its c-inseparable components.

To carry out step (1), we first construct the *containment graph* G^* of G, which is the directed graph with the same vertex set as G whose edges satisfy that $(u, v) \in G^*$ (directed from u to v) iff $N_G(u) \supseteq N_G(v)$ (u contains v in G). For each vertex u, it takes $O(m)$ time to determine all $N_G(v)s$ that contain $N_G(u)$. Hence, the construction of G^* takes $O(m \cdot n)$ time. Now, two vertices u and v are strictly adjacent iff $(u, v) \in G$ but there is no edge from one to the other in G^*; they are strongly adjacent iff $N_G(u) \cup N_G(v) = V$, $V \backslash N_G(v) = N_{G^*}(u)$ and $V \backslash N_G(u) = N_{G^*}(v)$. The partition of vertices in G_c not adjacent to a vertex v into L_v and R_v can be obtained from the graph G and its containment graph G^*.

If we assume that G satisfies (6.1), $V(G_c)$ must be a neighborhood module. Hence, in step (2), we first partition $V(G_c)$ into its maximal submodules using the substitution decomposition. Then refine each maximal series submodule M_i of $V(G_c)$ into its maximal consistent subsets $M^1, \ldots, M^r$ based on the following partitioning argument. Pick a vertex u in G_c, partition each maximal module M_i not containing u into $M_i \cap L_u$ and $M_i \cap R_u$ by scanning each vertex of L_u once. Now, iteratively refine the current partition based on the remaining vertices of G_c. Then the final partition of each M_i will be the collection of its maximal consistent subsets. This operation would take $O(m)$ time. Now, find the substitution decomposition tree of each M^j and combine them to form our consistent decomposition tree T. Hence, the total time spent in step (2) is $O(n^2)$ due to the substitution decomposition.

For step (3), the conformal models for series and parallel T-submodules can be easily composed from those of their children. To determine a conformal model for the representative graph G_M of each neighborhood T-submodule M (other than $V(G_c)$), we apply the join decomposition to G_M and combine the unique models of the j-inseparable components of G_M according to the algorithm described in the proof of Theorem 5.7. The conformal model for the representative graph of $V(G_c)$ is obtained using the above partitioning argument. These conformal models can then be used to compose a conformal model for T. In summary, the operations involved in step (3) are the substitution decomposition, the join decomposition, and the chord model construction in [5], each of which takes at most $O(m \cdot n)$ time.

The operations involved in the construction of T_{NS} in step (4) are very similar to the refinement procedure used in step (2). Pick a component C of G_c and a vertex u in C. Partition the remaining components into two clusters according to L_u and R_u. Iteratively refine the clusters based on every other vertex of C. At the end, each cluster consists of

components whose series-parallel relationships with respect to every vertex of C are identical. Furthermore, each cluster corresponds to a maximal NS-collection N_j containing C. This procedure can be recursively applied to the connected components within each cluster and we can eventually determine the series-parallel relationships of all components of G_c and construct the SP-tree T_{NS}. The total time complexity is $O(m+n)$.

Now consider step (5). Let G_i be a c-inseparable component graph of G_c. We shall follow the notations in §7.3 to construct a conformal model for G_i. Let C_i be the connected component of G_i with more than one vertices. Let K_i be the set of remaining vertices. The unique T-submodule $M(u)$ for each u in K_i can be determined in a bottom-up fashion. For each T-submodule M in the bottom level, we test which u in K_i splits M by keeping a vertex $u(M) \in C_i \backslash M$ and partition K with respect to each v in M according to L_v and R_v. Hence, all the augmented T-submodules and the consistent decomposition tree of G_c can be constructed in $O(m)$ time. Once the chord model for each c-inseparable component is available, composing a conformal model of G_c according to T_{NS} takes only $O(m)$ time.

9. The isomorphism problem on circular-arc graphs. We first consider the isomorphism problem for circular-arc graphs satisfying (6.1). To test the isomorphism between two such circular-arc graphs G and G', it suffices to test whether there exist isomorphic conformal models for G_c and G'_c. We shall make use of the unique consistent decomposition trees T and T' for G and G', respectively. A by-product of our algorithm is an isomorphism algorithm for circle graphs, which involves only the join decomposition. The unique consistent decomposition tree T of G_c satisfies the following properties. Its leaves are the vertices of G_c. It has a unique root and every interior node (except the root, which uses the consistent partition) is associated with N, P, or S, indicating three types of decompositions. The representative graph of an S-node is a clique. The representative graph of a P-node is an independent set (consisting of parallel vertices). The representative graph of an N-node is an s-inseparable component that is neither a clique nor an independent set. Each s-inseparable component of G_c corresponds to a uniquely representable induced subgraph of G. Since each internal node has at least two children, the total number of internal nodes (and therefore, the total number of nodes) of T is $O(n)$.

Define the *level* $\ell(v)$ of a node v in T to be the length of the unique path from v to the root. For each internal node v of T, denote the subtree with root v by T_v. Our isomorphism algorithm assigns integer labels to the nodes of T and T' such that a node v of T and another node v' of T' receive the same label iff there exists isomorphic conformal models for T_v and T'_v.

ISOMORPHISM ALGORITHM

(1) Determine the respective consistent decomposition trees T and T'. Assign the label 1 to each leaf of T and T'.

(2) Assign labels to the internal nodes of T and T' level iteratively with decreasing ℓ. Assume all nodes at level ℓ (denoted by S_ℓ, S'_ℓ, respectively) of T and T' have been assigned labels. Let v and v' be nodes in $S_{\ell-1}$ and $S'_{\ell-1}$, respectively.

(2.1) If v and v' correspond to series modules in T and T' and their children have the same collections of labels, then assign the same new label to them.

(2.2) If v and v' correspond to parallel modules in T and T' and the labels of their children follow the same unique parallel sequence, then they receive the same new label.

(2.3) If v and v' correspond to neighborhood modules (or, consistent partitions) in T and T', then there exist unique conformal models D and D', respectively, for their representative graphs H and H'. The isomorphism testing of v and v' can be reduced

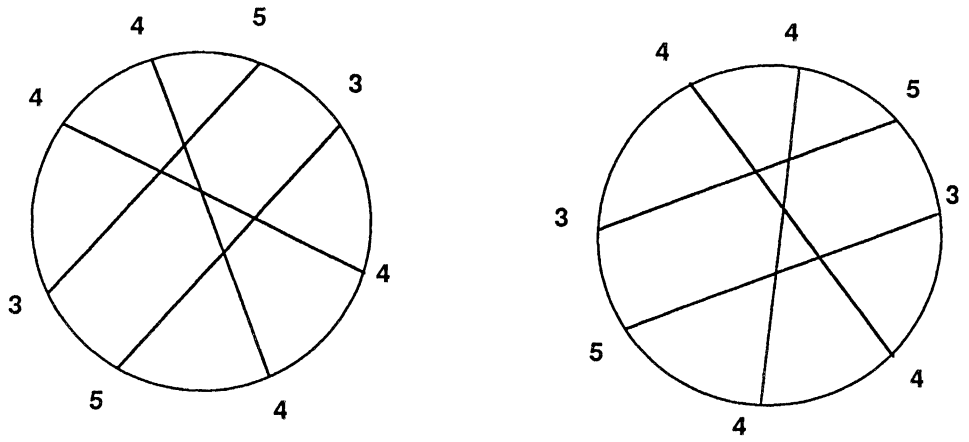


FIG. 9.1. *Testing the isomorphism of two unique chord models. If the chords are not labeled, then index each end the clockwise distance to its opposite end. Then, the isomorphism testing can be reduced to test if* 44534453 *is a substring of* 3445344534453445$3544354435443544. *If the chords are labeled, then in the above indexing scheme attach the label of the chord immediately after its index and apply string matching.*

to a string matching problem by using origin-free circular indexing of the endpoints of D and D' as shown in Fig. 9.1.

The time complexity for step (1) is $O(m \cdot n)$. Next, we claim the complexity for step (2) is $O(n^2)$ provided all the conformal models of the representative graphs are available. The labels assigned at each level starts from 0. Using bucket sort, we can obtain the sorted children label sequence of each node in level $\ell - 1$ in $|S_\ell|$ time. Step (2.1) can be done by comparing the sorted children label sequence. Step (2.2) can be carried out by comparing the unique label sequence. The complexity for (2.3) can be analyzed as follows. Testing the isomorphism between two unique conformal models D and D' as illustrated in Fig. 9.1 takes $O(|D|+|D'|)$ time. Hence, the time it takes to label all nodes in $S_{\ell-1}$ and $S'_{\ell-1}$ is $O(|S_\ell|^2)$. Let ℓ^* be the largest level number. Given the decomposition trees and conformal models for their representative graphs, the overall time complexity is bounded by $\sum_{\ell=1}^{\ell^*} O(|S_\ell|^2) = O(n^2)$, since $\sum_{\ell=1}^{\ell^*} |S_\ell| = O(|T|) = O(n)$.

Next, consider the isomorphism problem for circular-arc graphs whose associated G_c is not connected. Assume the SP-tree and all c-completion graphs have been computed. We divide our algorithm into two parts. First, we test their c-inseparable components for isomorphism. This can be carried out by testing the isomorphism of their corresponding (augmented) consistent decomposition tree using the algorithm in Fig. 9.1. Next, assign the same label to all components in the SP-trees T_{NS} and T'_{NS} of G_c and G'_c, respectively, if their c-inseparable components are isomorphic. Assign another distinct label to all nodes representing maximal NS-collections. Then the isomorphism testing of G_c and G'_c is reduced to that of two labeled trees T_{NS} and T'_{NS}.

Testing the isomorphism of two c-inseparable components, say H_c and H'_c, of G_c and G'_c takes $O(|V(H_c)|)$ time according to the previous analysis. Now, assigning labels to the components of the SP-trees can be done by testing for each component H_c of T_{NS} whose isomorphic components in T'_{NS} in $O(|V(H_c)| \cdot n)$ time. Hence, the total time for the label assignment of nodes in the SP-trees is $O(n^2)$. Finally, testing the isomorphism of two labeled trees takes time proportional to the number of vertices in the trees. Therefore, the total time spent in step (2) is $O(n^2)$.

Acknowledgments. We would like to thank the referees for their detailed comments that greatly enhanced the readability of this paper. We think a more careful implementation of the algorithm of this paper could reduce the complexity to $O(n^2)$. For example, a crucial step constructing the containment graph of circular-arc graphs in step (1) of §8 can be reduced to $O(n^2)$ time using a result of [4]. Since this result was submitted four years ago, some improvement has been obtained in the interim (notably, the result of [4]). However, we believe this paper has provided a useful framework for dealing with the structures of interval graphs and circular-arc graphs; in particular, the characterizations of prime components. Perhaps more efficient algorithms can be discovered along this line of thought.

REFERENCES

[1] K. S. Booth and G. S. Lueker, *Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms*, J. Comput. System Sci., 13 (1976), pp. 335–379.
[2] A Bouchet, *Reducing prime graphs and recognizing circle graphs*, Combinatorica, 7 (1987), pp. 243–254.
[3] W. H. Cunningham, *Decomposition of directed graphs*, SIAM J. Alg. Disc. Meth., 3 (1982), pp. 214–228.
[4] E. M. Eschen and J. Spinrad, *An $O(n^2)$ Algorithm for Circular-Arc Graphs*, Proc. of the 4th Annual ACM–SIAM Symposium on Discrete Algorithms, 1993.
[5] C. P. Gabor, W.-L. Hsu, and K. J. Supowit, *Recognizing circle graphs in polynomial time*, J. Assoc. Comput. Mach., 36 (1989), pp. 435–473.
[6] M. R. Garey, D. S. Johnson, G. L. Miller, and C. H. Papadimitriou, *The complexity of coloring circular-arcs and chords*, SIAM J. Alg. Disc. Meth., 1 (1980), pp. 216–227.
[7] F. Gavril, *Algorithms on circular-arc graphs*, Networks, 4 (1974), pp. 357–369.
[8] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.
[9] ———, *Interval graphs and related topics*, Discrete Math., 55 (1985), pp. 113–121.
[10] U. I. Gupta, D. T. Lee, and J. Y.-T. Leung, *Efficient algorithms for interval graphs and circular-arc graphs*, Networks, 12 (1982), pp. 459–467.
[11] W. L. Hsu, *Maximum weight clique algorithms for circular-arc graphs and circle graphs*, SIAM J. Comput., 14 (1985), pp. 224–231.
[12] V. Klee, *What are the intersection graphs of arcs in a circle?* Amer. Math. Monthly, 76 (1969), pp. 810–813.
[13] W. L. Hsu and K. H. Tsai, *Linear time algorithms on circular-arc graphs*, Inform. Process. Lett., 40 (1991), pp. 123–129.
[14] S. Masuda and K. Nakajima, *An optimal algorithm for finding a maximum independent set in circular-arc graphs*, SIAM J. Comput., 17 (1988), pp. 41–52.
[15] F. S. Roberts, *Graph Theory and Its Applications to Problems of Society*, Society for Industrial and Applied Mathematics, Philadelphia, 1978.
[16] J. Spinrad, *On comparability and permutation graphs*, SIAM J. Comput., 14 (1985), pp. 658–670.
[17] ———, *Circular-arc graphs with clique cover number two*, J. Combin. Theory Ser. B., 44 (1988), pp. 658–670.
[18] F. W. Stahl, *Circular genetic maps*, J. Cell Physiol., 70 (Suppl. 1) (1967), pp. 1–12.
[19] K. E. Stouffers, *Scheduling of traffic lights—A new approach*, Transportation Res., 2 (1968), pp. 199–234.
[20] W. Trotter and J. Moore, *Characterization problems for graphs, partially bordered sets, lattice, and families of sets*, Discrete Math., 16 (1976), pp. 361–381.
[21] A. Tucker, *Coloring a family of circular-arc graphs*, SIAM J. Appl. Math., 29 (1975), pp. 493–502.
[22] ———, *An efficient test for circular-arc graphs*, SIAM J. Comput., 9 (1980), pp. 1–24.
[23] T. H. Wu, *An $O(n^3)$ Isomorphism Test for Circular-Arc Graphs*, Dissertation, State University of New York, Stony Brook, 1983.

SIAM J. COMPUT.
Vol. 24, No. 3, pp. 440–456, June 1995

002

WHEN TREES COLLIDE: AN APPROXIMATION ALGORITHM FOR THE GENERALIZED STEINER PROBLEM ON NETWORKS*

AJIT AGRAWAL†, PHILIP KLEIN‡, AND R. RAVI§

Abstract. We give the first approximation algorithm for the *generalized network Steiner problem*, a problem in network design. An instance consists of a network with link-costs and, for each pair $\{i, j\}$ of nodes, an edge-connectivity requirement r_{ij}. The goal is to find a minimum-cost network using the available links and satisfying the requirements. Our algorithm outputs a solution whose cost is within $2\lceil \log_2(r+1) \rceil$ of optimal, where r is the highest requirement value. In the course of proving the performance guarantee, we prove a combinatorial min-max approximate equality relating minimum-cost networks to maximum packings of certain kinds of cuts. As a consequence of the proof of this theorem, we obtain an approximation algorithm for optimally packing these cuts; we show that this algorithm has application to estimating the reliability of a probabilistic network.

Key words. approximation algorithm, network design, Steiner tree problem

AMS subject classifications. 68R10, 68Q25

1. Introduction.

Consider the following scenario. Client industries of a telephone company have requested commercial telephone connections between pairs of their offices in different cities. The telephone company must then install a network of fiberoptic telephone links that accommodates all the clients' requirements. That is, the network must contain a path using these links between every pair of cities specified by the clients. Given the cost of installing links between different cities, the company must now decide which links to install so as to minimize its cost. (See Fig. 1.)

We formalize the problem as follows. Let G be a graph with nonnegative edge-costs, and let R be a set of node-pairs (s_i, t_i). We call these pairs *site-pairs*, and we say the nodes s_i, t_i are *sites*. We call a subgraph H of G a *requirement join* if H contains a path between s_i and t_i for every requirement (s_i, t_i). We call the node-pairs *requirements* because they represent connectivity constraints that must be satisfied by the output subgraph. We abbreviate *requirement join* by *R-join* when we want to emphasize the set R of requirements. The problem we consider in this paper is to find a minimum-cost R-join.

The problem faced by the telephone company can be directly formulated as a minimum-cost R-join problem. In this formulation, it is assumed that a link can be used simultaneously by many clients. This assumption is reasonable in light of the very high bandwidth of fiberoptic links.

Consider the special case of this problem in which there is a set T of *terminals*, and every pair of nodes in T needs to be connected. This special case is known in the literature as the *Steiner tree problem in networks*. This problem was one of the first seven problems shown to be NP-complete by Karp [19]. Given the range of its applications, it is not surprising that this problem has been well-studied. Many enumeration algorithms, heuristics [33], [44], [18], and approximation algorithms [6], [38], [25], [11], [31], [37], [29], [45] are known for the problem. Polynomial-time solutions for restricted classes of graphs are also known (see [42]).

*Received by the editors August 24, 1992; accepted for publication (in revised form) October 7, 1993. A preliminary version of this paper appeared in the *Proceedings of the* 23rd *Annual ACM Symposium on Theory of Computing* (1991), pp. 134–144. This research was supported by National Science Foundation grant CCR-9012357, National Science Foundation grant CDA 8722809, Office of Naval Research and DARPA contract N00014-83-K-0146 and ARPA Order No. 6320, Amendment 1.

†Exa Corporation, Cambridge, Massachusetts, 02140 (agrawal@exa.com).

‡Brown University, Department of Computer Science, Box 1910, Providence, Rhode Island 02912 (pnk@cs.brown.edu).

§University of California, Department of Computer Science, Davis, California 95616 (ravi@cs.ucdavis.edu).

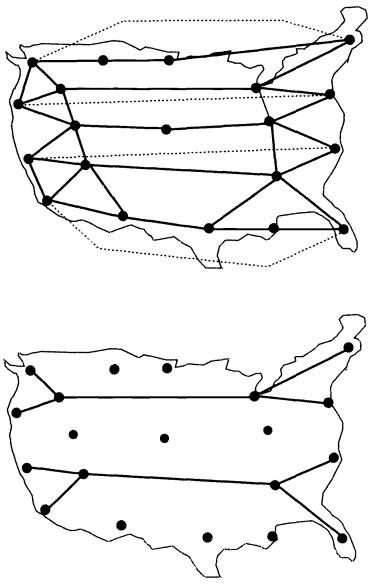


FIG. 1. *An instance of the unweighted network design problem and its solution. Solid edges correspond to unit-cost links; dotted edges connect site pairs.*

However, none of the algorithms addresses the more general case in which each client can specify an arbitrary pair of cities. Note that in this general case the solution network need not be connected. Moreover, a minimum-cost Steiner tree solution can be arbitrarily costlier than a minimum-cost solution to this more general problem.

In this paper, we give the first approximation algorithm for the minimum-cost R-join problem.

THEOREM 1.1. *There is a polynomial-time algorithm for finding an R-join of cost at most $2 - 2/k$ times minimum, where k is the number of sites.*

1.1. The generalized Steiner problem in networks. The algorithm of Theorem 1.1 is useful when the network to be constructed need not be connected. However, the algorithm is also useful, as a subroutine, even in designing connected networks. Namely, we consider a generalization of the minimum-cost R-join problem involving certain redundancy requirements.

Consider the scenario described above but where each client can specify that her pair of cities must be connected by some number of edge-disjoint paths so that the connection is less vulnerable to link failure. The goal is to design a network satisfying these specifications. The network is allowed to contain multiple links between the same pair of nodes; all such links have the same cost.

To model this situation, we allow more general requirements. The set R of requirements consists of triples (s_i, t_i, r_i), where r_i, the requirement value, is a positive integer. An *R-multijoin* is a multiset of the edges of G that contains r_i edge-disjoint paths from s_i to t_i, for every requirement (s_i, t_i, r_i). The cost of an R-multijoin is the sum of the costs of the edges in the multiset, counting multiplicities. Using our approximation algorithm for minimum-cost R-join, we obtain an approximation algorithm for minimum-cost R-multijoin.

THEOREM 1.2. *There is a polynomial-time algorithm for finding an R-multijoin of cost at most $(2 - 2/k)\lceil \log_2(r_{\max} + 1)\rceil$ times minimum, where $r_{\max}$ is the largest requirement value and k is the number of sites.*

This problem is called *multiterminal network synthesis* by Chien [7] and Gomory and Hu [13]. Gomory and Hu and later Sridhar and Chandrasekaran [35] address the synthesis problem for the special case where the input graph is the complete graph with all costs identical.

The problem is also essentially identical to the *generalized Steiner problem* as formulated by Krarup ([26], as cited in [41]). The problem is referred to as the design of minimum-cost *survivable networks* in the work of Steiglitz, Weiner, and Kleitman [36]. These researchers pose the problem of finding a subgraph of minimum cost satisfying given connectivity requirements. The problem we address differs in that we allow the solution to contain multiple copies of links appearing only once in the input graph G.

In this paper we will use the term *R-multijoin* whenever the requirements R include requirement values exceeding one, and will reserve the term *R-join* for the case when the requirement values are all one. The former case is addressed only in §3.3, where we show how to reduce the problem to the latter case. The remainder of the paper addresses only the latter case.

1.2. Packing cuts, with application to network reliability. Our results also have application to evaluating network reliability. Suppose the telephone company has an existing network and the same list of clients, each specifying a pair of cities. The company needs to determine how likely it is that random failure of communication links renders some of its clients' requirements unsatisfiable.[1] Assuming link failures are independent, determining the probability that the surviving links can serve all clients' requirements is a generalization of the #P-complete problem [39] called network reliability. No approximation algorithms are known.

However, one powerful and useful heuristic for estimating two-terminal and k-terminal reliability [8], [9] can be directly generalized to handle the case of arbitrary pairs. The (generalized) heuristic consists in finding a large collection of edge-disjoint cuts in the network such that each cut separates at least one client's pair of cities. For a surviving network to be able to serve all clients' requirements, at least one edge in each cut must survive; thus such a cut-packing can be used to obtain a lower bound on the probability of catastrophic failure. Experience [9] with this heuristic in the cases of two-terminal and k-terminal reliability indicates that it is one of the best available.

One of the results of this paper is an algorithm for finding a nearly maximum collection of such cuts in an auxiliary network whose reliability is the same as that of the original network. We give more details in §3.

1.3. The combinatorial basis for our algorithms: A new approximate min-max equality. At the heart of our proofs of near-optimality is a combinatorial theorem that relates the R-join problem to the cut-packing problem in the case of unit edge-weights.

THEOREM 1.3. *The minimum size of an R-join is approximately equal to one-half the maximum size of a collection of cuts, where each cut separates some site-pair, and no edge is in more than two cuts. By "approximately," we mean within a factor of* $2 - 2/k$, *where* k *is the number of sites.*

The proof of Theorem 1.3 is algorithmic and is given in §5. We can formulate the two combinatorial quantities as the values of integer linear programs that are dual to one another. It follows from Theorem 1.3 that the fractional relaxations of these programs provide good approximations to both combinatorial quantities. Moreover, the factor of $2 - 2/k$ is

[1] A related problem—finding the minimum number of communication links that would need to fail for *all* requirements to be unsatisfiable—can be solved approximately, using techniques we presented in an earlier paper [22].

existentially tight, as shown by the example of a k-cycle given by Goemans and Bertsimas [14].

2. Related work.

2.1. The Steiner tree problem in networks. There have been volumes of work done on the Steiner tree problem in networks, including proposed solution methods, computational experiments, heuristics, probabilistic and worst-case analyses, and algorithms for special classes of graphs. Winter [41] and more recently Hwang and Richards [17] surveyed this body of work.

Karp [19] showed that the problem is NP-complete. Takahashi and Matsuyama [38], Kou, Markowsky and Berman [25], El-Arbi [11], Rayward-Smith [33], Aneja [2], and Wong [44] are among those who proposed heuristics. Among these, the heuristics that have been analyzed have a worst-case performance ratio of $2 - 2/k$, where k is the number of terminals that need to be connected (called Z-vertices in [41]). One algorithm, proposed by Plesnik [31] and by Sullivan [37], performs somewhat better. Recently Zelikovsky gave an approximation algorithm with a performance ratio of 11/6 [45]. Berman and Ramaiyer [6] have improved this to 16/9.

In computational experiments, these heuristics generally perform considerably better than the worst-case bounds. Jain [18] proposed an integer-program formulation of the Steiner tree problem in networks, and showed that for two random distributions of costs, the value of this integer program differed drastically from the value of its fractional relaxation.

2.2. The generalized Steiner problem in networks. The *generalized Steiner problem in networks*, as originally formulated by Krarup (see [42]), is as follows. The input consists of a graph with edge-costs, a subset Z of the vertices, and, for each pair of vertices $i, j \in Z$, a required edge-connectivity r_{ij}. The goal is to output a minimum-cost subnetwork satisfying the connectivity requirements. When the r_{ij}'s are allowed to be zero, we can clearly assume without loss of generality that Z consists of all the vertices of the graph.

Previous to our work, no approximation algorithms for the generalized Steiner problem were known. There have been papers on finding exact solutions and on algorithms for special classes of graphs [41], [42].

In the work of Goemans and Bertsimas, described below, and in our work, the edge-connectivity requirement is allowed to be satisfied in part by duplicating edges of the input graph. This corresponds to "buying" multiple communication links of the same cost and with the same endpoints.

2.3. Survivable networks. In recent work, Goemans and Bertsimas [14] considered a special case of the generalized Steiner problem in networks. Instead of arbitrary requirement values, the input includes an assignment of integers r_i to nodes. The goal is to find a minimum-cost network satisfying requirements $r_{ij} = \min(r_i, r_j)$. They propose a simple but powerful approach which involves solving a series of ordinary Steiner tree problems using a standard heuristic. They show that this approach yields solutions that are within a factor of $2\min(\log R, p)$ of optimal, where R is the maximum r_i and p is the number of distinct nonzero values r_i in the input. Moreover, they show that their analysis is tight in the worst case.

Goemans and Bertsimas restricted their attention to edge-connectivity requirements of the special form $r_{ij} = \min(r_i, r_j)$ so that each subproblem has essentially the form of an (ungeneralized) Steiner tree problem. That enabled them to solve each subproblem approximately, using one of the known approximation algorithms for the Steiner tree. By providing an approximation algorithm for the case of $r_{ij} \in \{0, 1\}$, we make it possible to handle requirements r_{ij} not of that special form.

2.4. Subsequent work. Building on our result, Goemans and Williamson [16] simplified and generalized our algorithm. They describe a framework in which to formulate and find approximately optimal solutions for many *constrained forest problems*, of which the minimum-cost R-join problem is an example. Their approximation algorithm uses an approach similar to ours and achieves the same performance guarantee. Goemans and Williamson describe an implementation of their algorithm that runs in $O(n^2 \log n)$ time on graphs with n nodes.

In work building on that of Goemans and Williamson, we showed [32] how to obtain approximately optimal solutions to 2-edge-connected versions of the problems addressed in [16]. In these problems, one needs to achieve 2-connectivity without duplicating links. Finally, several subsequent papers [15], [23], [40] extended these methods to give approximation algorithms for the generalized Steiner problem without link duplication.

3. Background. An instance of the generalized Steiner problem consists of a graph G with edge-costs c, together with a collection $\{R_1, \ldots, R_b\}$ of *requirements*: each requirement R_i consists of a *site pair* $\{s_i, t_i\}$, a pair of nodes of G, and a *requirement value* r_i, a positive integer. A feasible network is a multiset N consisting of edges of G, such that for every requirement $R_i = (\{s_i, t_i\}, r_i)$, there are at least r_i edge-disjoint paths between s_i to t_i in the multigraph with edges N.

3.1. The unweighted case. To prove performance guarantees for our algorithm, we exploit an approximate duality between feasible networks and packings of cuts. Fix some instance of the generalized Steiner problem, where all costs and requirement values are 1. Thus the instance consists of a graph G and a collection of site pairs $\{s_i, t_i\}$. Let k denote the cardinality of the set of sites, i.e., the set of nodes appearing in site pairs. Note that the number of sites may be significantly smaller than the total number of nodes. A *feasible network* is a subgraph in which, for every site pair $\{s_i, t_i\}$, there is a path between s_i and t_i.

Let N be any feasible solution for this instance. Observe that if N is minimal, then it is just a forest. Let S be any subset of nodes of G such that for some site pair $\{s_i, t_i\}$, one of the sites is in S and one is not. In this case, the set of edges A with exactly one endpoint in S is called a *requirement cut*. There must be a path between s_i and t_i in N, so N intersects A in at least one edge. Thus we have the following lemma.

LEMMA 3.1. *Every feasible network and every requirement cut have at least one edge in common.*

Suppose $A_1, \ldots, A_\beta$ are (not necessarily distinct) requirement cuts such that each edge of G occurs in at most two cuts. We call such a collection of cuts a 2-packing. Then we have the following easy lower bound on the minimum size of a network design.

LEMMA 3.2. *The minimum size of a feasible network is at least one-half the maximum size of a 2-packing of requirement cuts.*

Proof. Let N be a feasible network and let $A_1, \ldots, A_\beta$ be a 2-packing consisting of β requirement cuts. We have

$$|N| \geq \sum_{e \in N} \frac{1}{2} |\{i : e \in A_i\}| = \frac{1}{2} \sum_{i=1}^{\beta} |A_i \cap N| \geq \frac{1}{2}\beta \tag{1}$$

because each $|A_i \cap N|$ is at least one. □

For comparison, Edmonds and Johnson [10] show that T-joins and T-cuts satisfy an analogous inequality, and, more importantly, they satisfy it with equality.

Instead of showing equality, we show approximate equality, to within a factor of $2(1 - 1/k)$. This is the content of Theorem 1.3.

Our proof of Theorem 1.3 is algorithmic. We give an algorithm that constructs a feasible network and a 2-packing, such that the first has size at most $(1 - 1/k)$ times the second. It fol-

lows that the feasible network is approximately minimum and the 2-packing is approximately maximum, to within a factor of $2(1 - 1/k)$.

The first step is to transform the original graph G_0 into a bipartite graph G by replacing each edge uv of G_0 with two edges ux and xv in series, where x is a new node. The resulting graph G has the following properties:

- Any minimal feasible network in G corresponds to a feasible network in G_0 of half the size.
- Any packing of edge-disjoint requirement cuts in G corresponds to a 2-packing of requirement cuts in G_0 of the same size.

Consequently, in order to prove Theorem 1.3 for G_0, it is sufficient to show the following for G:

(2) we can find a feasible network N and a packing of edge-disjoint requirement cuts $A_1, \ldots, A_\beta$ such that $N \leq 2(1 - 1/k)\beta$, where k is the total number of sites.

We show (2) in §§4 and 5.

3.2. The weighted case. Now we consider the case in which the costs of edges may vary, but the requirement values are still all one. It turns out that, like Edmonds and Johnson's theorem, Lemma 3.2 and Theorem 1.3 are self-refining. For nonnegative integer edge-costs c, we simply replace each edge e by a path of length $c(e)$. We say a collection of requirement cuts is a $2c$-packing if each edge e appears at most $2c(e)$ times. Using this transformation, we obtain the following theorem from Theorem 1.3.

THEOREM 3.3. *The minimum-cost of a feasible network is at least one-half the size of a $2c$-packing of requirement cuts, and at most $(1 - 1/k)$ times this size.*

To actually compute an approximately minimum feasible network, we use a more direct approach, which we describe in §4.2.1.

3.3. Arbitrary integral requirements. So far we have dealt with the case in which each site pair need only be connected in the final feasible network. As discussed in the introduction and §2, a client may also require that there be at least r_{ij} edge-disjoint paths between her pair of sites.[2] Thus the case dealt with up to now requires each r_{ij} to be either 0 or 1.

In order to obtain an approximation algorithm for this generalized problem from our algorithm for the case of 0–1 requirements, we make use of a heuristic technique due to Goemans and Bertsimas [14]. They propose a technique they call the *tree heuristic*, which consists essentially of decomposing a problem with many different requirement values into a series of simpler problems in which only two requirement values appear. As we mentioned in §2, they use the technique for solving only a special case of the generalized Steiner problem. In conjunction with our new algorithms for the 0–1 case, however, the technique can be easily adapted to apply to the general case.

Let the different values of r_{ij} be $0 = p_0 < p_1 < p_2 < \cdots < p_s$. For each $0 < d \leq s$, consider the transformed problem

$$r_{ij}^d = \begin{cases} p_d - p_{d-1} & \text{if } r_{ij} \geq p_d, \\ 0 & \text{otherwise,} \end{cases}$$

which is essentially $p_d - p_{d-1}$ copies of a 0–1 problem. Use a standard heuristic to find an approximately optimal solution, and combine the solutions to the s transformed problems to

[2] In this case, the feasbile network is allowed to use multiple copies of edges of the input graph; each copy of a given edge costs the same.

get a solution to the original problem. The resulting performance guarantee is $\Theta(s)$.[3] By using a similar approach, if each r_i is an integer b bits long, then the original problem can be decomposed into b problems, and the resulting performance bound is $2(1 - 1/k)b$. This is how we get the performance bound stated in Theorem 1.2.

The obvious question is whether one can do better than this. Goemans and Bertsimas can show that their analysis is tight, so another approach is needed, one that can deal simultaneously with widely varying requirement values.

3.4. Reliability estimation. In the introduction, we described a heuristic for estimation of network reliability in a probabilistic network. In order to use this heuristic effectively, we want to find a maximum collection of edge-disjoint requirement cuts. This problem is NP-complete for general graphs. Moreover, an approximation algorithm for this cut-packing problem would yield an approximation algorithm for maximum independent set [9], an unlikely outcome in view of recent results [3], [4], [12]. We instead show how to make use of a cut-packing in bipartite graphs. We apply the transformation described in §3.1 to turn an arbitrary graph into a bipartite graph with all sites on one side of the bipartition: replace an edge having failure probability $1 - p$ with two series edges each having failure probability $1 - \sqrt{p}$. We do not change the probability of reliability in carrying out this transformation, and we can apply the algorithm of §4 to find an approximately maximum set of edge-disjoint cuts in the resulting graph.

Thus we propose a four-step recipe for estimating network reliability. Transform the network into a bipartite network, find an approximately maximum cut-packing, compute for each cut the probability that at least one edge survives, and multiply these probabilities to get an upper bound on the probability that all clients can continue to communicate.

4. The algorithm. In this section, we describe an algorithm for finding a cut-packing and an R-join. In §4.2, we describe how to find a cut-packing in the case of unit edge-weights. In §4.2.1 we describe the modification needed to handle arbitrary edge-weights. The algorithm for finding an R-join is the same in the two cases.

4.1. Overview. We start by providing an overview of the algorithm for the case of unit edge-weights. The algorithm grows breadth-first search trees from the sites, accumulating cuts as it proceeds. The algorithm employs a notion of timesteps. At each timestep, each of the breadth-first trees grows by an additional level. Each tree grows until all the sites it contains have found their mates. When trees collide, they are merged. As the algorithm grows trees, it builds networks spanning the sites in each of these trees. Using a charging scheme, we show that the size of each network in a tree is about twice the number of cuts accumulated while growing the tree.

4.2. Finding a cut-packing. Assume the input graph has unit edge-weights; we briefly address the more general case at the end of this subsection. Let G be a bipartite graph with all sites on the same side of the bipartition. (We can obtain such a graph from an arbitrary graph as described in §3.1. All subsequent references to the "original graph" refer to G.) We are given a collection of site pairs $\{s_1, t_1\}, \{s_2, t_2\}, \ldots, \{s_b, t_b\}$. We refer to the nodes s_i, t_i as *sites*. We say that two sites in the same site pair are *mates* of each other.

The algorithm for constructing the cut-packing is quite intuitive. (A summary is given at the end of this subsection.) We grow disjoint breadth-first search trees from all sites s simultaneously. We call the edges connecting one level to the next in a breadth-first search tree a *level cut*. Each level cut in a breadth-first search tree rooted at s is a requirement

[3] More specifically, Goemans and Bertsimas show the performance bound is $2(1 - 1/k)(\sum_{d=1}^{s}(p_d - p_{d-1})/p_d)$.

cut because its edges separate s from its mate. Thus at each timestep, we accumulate one additional requirement cut for each tree being grown.

When multiple trees collide, we merge them into a single tree and continue growing from its boundary. Thus in general a tree may contain many sites. As soon as every site in a tree has its mate in the same tree, we can no longer guarantee that subsequent level cuts of the tree are requirement cuts, so we call the tree *inactive*, and we contract all its nodes into a single *supernode*. A tree that is still in the process of being grown is said to be *active*. The algorithm terminates when there are no active trees. At this point, every site pair's two nodes are contained in the same tree. More precisely, since each tree has become inactive, and has hence been contracted to a supernode, there are no sites remaining in the graph.

Because of contractions, the graph on which we are working evolves during the course of the algorithm. We use G_t to denote the graph after t timesteps. When we refer to a graph, unless we explicitly call it the "original graph," we will mean the contracted graph G_t at a certain point t in the algorithm.

It is important to the analysis that all active trees grow at the same rate. The algorithm takes place over a series of timesteps. In each timestep, each active tree grows by one level. Thus after t timesteps, active trees that have not participated in any collisions all have radius t (as measured in the contracted graph G_t). More generally, let the *boundary* of a tree be the set of nodes at the most recent level of the tree. We have the following proposition.

PROPOSITION 4.1. *After t timesteps, each node in the boundary of an active tree is distance t from some site internal to the tree.*

In the initial bipartite graph, all the sites are on the same side of the bipartition. We show that this property continues to hold throughout the algorithm.

LEMMA 4.2. *After t timesteps, the graph is still bipartite, with all sites on the same side of the bipartition.*

Proof. The proof is by induction on t. The basis $t = 0$ is trivial. We must show that the bipartition property described in the lemma is preserved by contractions. Suppose that G_{t-1} obeys the property, and that after t timesteps, some tree T has just become inactive and is about to be contracted. By Proposition 4.1, all the nodes in the boundary of T have distance t from some site. Hence they all belong in the same side of G_{t-1}'s bipartition. It follows that after the nodes of T are contracted to a single node, the bipartition property still holds. □

We can use Lemma 4.2 to show that all the cuts found by the algorithm are edge-disjoint.

COROLLARY 4.3. *No edge belongs to a level cut of more than one tree.*

Proof. By Proposition 4.1 and Lemma 4.2, all the nodes in boundaries of all active trees are in the same side of the bipartition of the graph. Hence no edge is incident to two active trees. □

Thus trees collide by reaching the same node in a given step. Below we summarize the cut-packing algorithm. In anticipation of the analysis of the algorithm, we "assign" cuts found to particular trees.

1 Initialize each site to be an active tree. Repeat the following steps until every tree is inactive.
2 Grow each tree by one level. Assign the corresponding level cut to the tree.
3 Contract each tree that has just become inactive.
4 Repeat
5 Take two distinct trees sharing a boundary node, and merge them into a single tree. (For the cut-packing algorithm, merging trees consists merely of taking the union of their nodes and of the cuts assigned to them.)
6 Until no more trees can be merged.

Because trees are merged immediately after they collide, we can claim the following. Just before the trees are grown, they are node-disjoint. Just after the trees are grown, they are *internally* node-disjoint: only their boundaries can share nodes. We make use of this property in the next subsection.

4.2.1. The cut-packing algorithm for weighted edges. The cut-packing algorithm in the case of weighted edges is only slightly more elaborate. We describe it in this subsection. The algorithm to find a feasible network based on the cut-packing, described in the next section, remains unchanged.

The key is to carry out many timesteps in a single iteration. It is useful to imagine that in each iteration, the growing trees continuously "consume" all their incident edges at the same rate until some edge is completely consumed, at which time things must be updated. We assume for simplicity that the edge-weights are integral. For each edge, we maintain a variable indicating how much of that edge remains to be consumed. To determine the amount λ by which to grow active trees in an iteration, we compute two minima:

$$\lambda_1 = \min_e \text{ amount of } e \text{ yet to be consumed,} \tag{3}$$

where the min is over edges e that have one endpoint in an active tree, and

$$\lambda_2 = \min_e \text{ amount of } e \text{ yet to be consumed,} \tag{4}$$

where the min is over edges e that have both endpoints in distinct active tree.

Finally, we let $\lambda = \min\{\lambda_1, \frac{1}{2}\lambda_2\}$.

To grow the trees by λ, we update the variables associated with edges: each edge having one endpoint in an active tree has its variable decreased by λ, and each edge having its endpoints in distinct active trees has its variable decreased by 2λ (because each such edge is being consumed from both sides). Then we execute steps 3 through 6 of the unweighted algorithm. It follows by the definition of λ that at least one edge is wholly consumed in an iteration, hence at least one tree grows by at least one node. For a tree T, let t_T be the number of nodes in T. It follows that the potential function $\sum_T t_T$ − (number of trees) goes up by at least one in each iteration, and hence that the number of iterations is at most the number of nodes in the graph. Thus the cut-packing algorithm requires only polynomial time.[4]

4.3. The network-design algorithm. The basic approach to building a feasible network is also quite intuitive. For each tree, we maintain a connected network connecting together all sites in the tree. This is easy: start with each site being a network in itself, and, whenever trees merge, use simple paths to join up their two networks.

It is possible to show that for each tree, the size of a network for that tree is no more than twice the number of cuts assigned to the tree. Such an analysis, however, is insufficient: the networks formed in this way are not connected in the original graph, because of the contractions we have performed along the way. A path that contains a supernode is not in general a path in the original graph. Therefore, we must be more careful in joining networks, and must not forget to include edges between nodes within inactive trees. Note that such edges do not even appear in the contracted graph G_t.

We introduce some terminology to help us relate various contracted graphs to each other and to the original graph. We call a node a *real node* if it appears in the original graph, in

[4]Using a heap to organize the edges incident to each tree, one can implement the algorithm to run in $O(n^2 \log n)$ time [1]. Using a more sophisticated two-level heap structure, one can implement it in $O(n\sqrt{m} \log n)$ time [21].

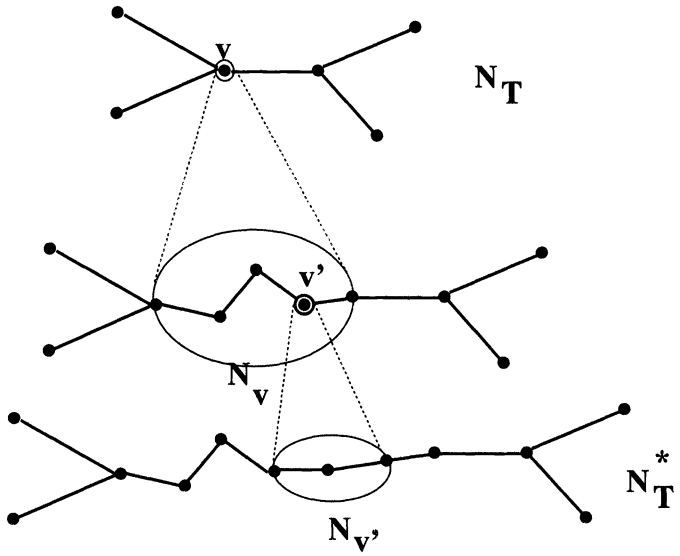


FIG. 2. *The network N_T corresponds in a natural way to a subgraph N_T^* of the original graph. To obtain N_T^* from N_T, replace each supernode v in N_T with the subgraph N_v and recurse on the supernodes in N_v, like v' above.*

order to distinguish such nodes from supernodes. If the tree T was contracted to form the supernode v, we say T *corresponds* to v, and vice versa. We say v *immediately encloses* v' if v is a supernode corresponding to a tree T containing v'. Note that each node is immediately enclosed by at most one node. A node enclosed by another node does not appear in the current graph, but we cannot simply forget about it since it continues to play a role in the algorithm.

We define the relation *encloses* to be the reflexive and transitive closure of the relation *immediately encloses*. That is, v encloses v' if by some series of contractions, v' was identified with other nodes to form v.

For an edge e incident to a node v in a contracted graph, there is a real node v' enclosed in v such that e is incident to v' in the original graph. We say that v' is the real node *by which e is incident to v*.

For each tree T, we maintain a *network* N_T, a subgraph of T. We maintain the following *site-inclusion invariant*:

- For each T, the network N_T includes all sites that are nodes of T.

We specifically mean to exclude those sites strictly enclosed by supernodes belonging to T. The site-inclusion invariant speaks only of those sites that are themselves nodes of T. If v is a supernode corresponding to an (inactive) tree T, we use T_v to denote T, and we use N_v to denote N_T. We say a node is *free* if it is not contained in any network N_T.

Each network N_T corresponds in a natural way to a subgraph N_T^* of the original graph. Namely, to get N_T^* from N_T, replace each supernode v in N_T with the subgraph N_v, and recurse on the supernodes in N_v (see Fig. 2).

We want each network N_T to correspond to a connected subgraph in the original graph. We therefore maintain the following *connectivity invariant*:

- Each subgraph N_T^* is connected.

At any stage in the algorithm, the networks N_T induce a subgraph of the original graph, namely the subgraph induced by the edges in $\bigcup_T N_T$ where the union is over all trees active and inactive. Let us call this subgraph N. Note that each induced subgraph N_T^* is a subgraph of N.

We now observe that when the algorithm terminates, the invariants imply that N is indeed a feasible network—for each site pair $\{s_1, s_2\}$, there is a path in N between s_1 and s_2. Let T be the tree containing s_1. Once the algorithm terminates, T must be inactive, and hence contains s_1's mate s_2 as well. By the site-inclusion invariant, s_1 and s_2 are nodes of N_T. Since they are original nodes, they are also nodes of the induced subgraph N_T^*, which is a subgraph of N. Finally, by the connectivity invariant, N_T^* is connected, so the required path exists.

Now we give the algorithm for network design. We run the cut-packing algorithm of the last subsection and, whenever a "merge" of trees occurs, we update the N_T's in order to maintain the invariants. Initially, when every active tree T consists of a single site, N_T consists also of this site. For each tree T not yet formed, N_T is empty. Thus trivially the invariants hold initially.

In step 5 of the cut-packing algorithm, we merge a pair of distinct trees T_1 and T_2 sharing a common boundary node v. By simply taking the union of their networks N_{T_i}, we get a network that obeys the site-inclusion invariant. However, this network does not obey the connectivity invariant. We must therefore connect up these networks. To do this, we add paths from the common node v to each of the networks N_{T_i}. This involves some care when v is a supernode. However, in this description of the algorithm, we postpone discussion of this case until §4.4. Assume therefore that v is a real node. We call a procedure CONNECTTONETWORK (v, T_i) for $i = 1, 2$.

The goal of CONNECTTONETWORK (v, T) is to augment various networks $N_{T'}$ until v is connected to N_T^*. To do this, the procedure first finds a shortest path P_0 in T from v to a site in T, identifies the shortest initial subpath P of P_0 that ends on a node of N_T, and adds the edges of P to N_T. We are not done yet; P does not necessarily correspond to a connected subgraph of the original graph because it may contain supernodes. Moreover, we have just added such supernodes u to N_T, so the networks N_u corresponding to these supernodes belong to N_T^*. In order to maintain the connectivity invariant, therefore, we must connect the networks N_u to N_T^*. We make these connections recursively using a procedure EXPANDPATH (u, P). This procedure expands P into a real path (i.e. a path in the original graph) by replacing each supernode u in the path with a subpath within T_u that connects a boundary node of T_u to u's network, goes through that network, and comes out again to the boundary of T_u. For technical reasons, EXPANDPATH does not replace the last node of P, so if this last node is a supernode, we use a recursive call to CONNECTTONETWORK to make this part of the path real. Making a path real using EXPANDPATH and CONNECTTONETWORK is illustrated in Fig. 3.

Now we give the procedure for CONNECTTONETWORK (v, T). Once again, the basic idea is to find a short path P in T from v to the network N_T, then introduce additional edges to make P correspond to a real path, i.e., a path among the real nodes.

CONNECTTONETWORK(v, T)

Assumption: The node v is a real node enclosed by some node v_0 in the boundary of T.

C1 Let v_0 be the node in T that encloses v.

C2 Let P_0 be a shortest path in T from v_0 to a site s. Let v_r be the first node of P_0 belonging to N_T, and let P be the subpath of P_0 from v_0 to v_r.

C3 Add P to N_T.

C4 Call EXPANDPATH(v, P) to make a real path out of P, except possibly for the last connection.

C5 If the last node v_r in P is a real node, then stop.

C6 Else,

C7 Let T' be the (inactive) tree corresponding to the supernode v_r.

C8 Let v' be the real node by which the last edge of P is incident to v_r.

C9 Recursively call CONNECTTONETWORK(v', T').

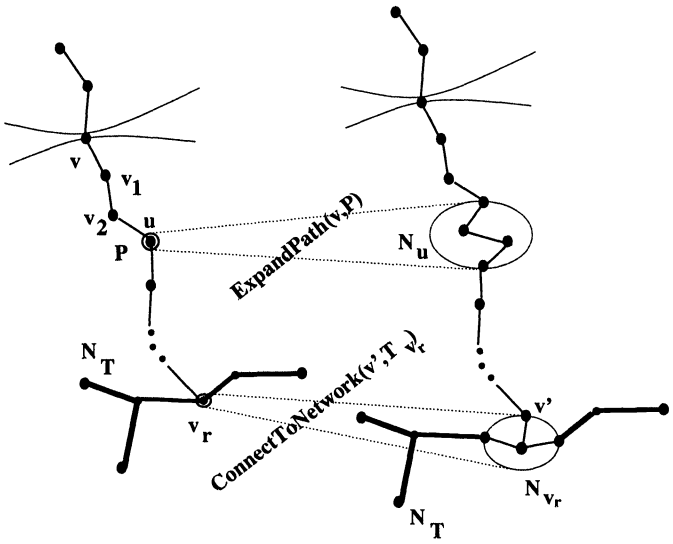


FIG. 3. *After identifying a path P of $v_1, \ldots, v_r$ to connect v to T, any supernode u in P is expanded by* EXPANDPATH(v, P) *into a connection via the network tree of u, N_u. If the last node in P, v_r is a supernode, it is recursively expanded to a path in the original graph by identifying a vertex v' on its boundary and calling* CONNECTTONETWORK(v', T_{v_r}) *recursively to connect v' to the network N_{v_r}.*

The procedure CONNECTTONETWORK uses a subprocedure EXPANDPATH(v, P) to make a real path out of P. For each node v of P except the last, if v is a supernode, we may have to add edges to N_v.

EXPANDPATH(v, P)
Assumption: P is a path in some tree T, whose first node encloses v, which is assumed to be a real node.

```
E1    Write P = v0 e0 v1 e1 ... e_{r-1} v_r.
E2    For i := 0 to r - 1 do
E3      Let v' be the real node by which e_i is incident to v_i.
E4      Comment: We must make a real path in N from v to v'.
E5      If v_i is a supernode then
E6        Let T be the tree corresponding to v_i.
E7        Call CONNECTTONETWORK(v, T).
E8        Call CONNECTTONETWORK(v', T).
E9        Comment: Now there is a real path from v to T's network to v'.
E10     Let v be the real node by which e_i is incident to v_{i+1}.
```

To prove that by using these procedures in the merge we maintain the connectivity invariant, we would use induction to show the following two statements. The call CONNECTTONETWORK(v, T) introduces edges in N_T^* to connect the real node v to N_T^*. The call EXPANDPATH(v, P) introduces edges in the networks N_{v_i} (for each supernode $v_i \in P$ except the last) so that the edges of P are connected up in N.

4.4. Merging trees whose common node is a supernode. To complete the description of the algorithm, we consider the case in which the node at which trees collide is a supernode rather than a real node. Let v be a supernode and suppose trees $T_1, \ldots, T_k$ collide at v at time t. We describe how to merge these trees.

To initialize, let $T = T_1$. For $i = 2$ to k, we merge T_i into T as follows. Since v is on the boundary of T_i, Proposition 4.1 ensures a path of length t from v to a site of T_i. Let e be the first edge on this path and let v_i be the real node by which e is incident to v. We then call CONNECTTONETWORK(v_i, T) and CONNECTTONETWORK(v_i, T_i). These calls establish a path going through v_i between the network of T and the network of T_i. We then let T be the resulting merged tree, i.e., $T := T \cup T_i$. This completes the merge of T_i into T.

The second invocation, CONNECTTONETWORK(v_i, T_i), needs some elaboration. As we shall see in the next section, the analysis of the algorithm requires that steps E7 and E8 of EXPANDPATH be executed at most once for a given tree T during the course of the algorithm. The first invocation of CONNECTTONETWORK(v_i, T) executes these two steps for the tree T_v corresponding to the supernode v. We must therefore avoid executing these steps in subsequent invocations of CONNECTTONETWORK. Fortunately, the choice of v_i enables us to avoid executing these steps, as we now explain.

Step C2 of CONNECTTONETWORK selects a path P_0 from the supernode v to a site in T_i. By choice of v_i, we can select the path P_0 so that its first edge is incident to the real node v_i in G. P is an initial subpath of P_0. Therefore, when we call EXPANDPATH(v_i, P) in step C4, we omit the iteration $i = 0$ in EXPANDPATH in which P's connection to v_i is made a real connection. This omission avoids reexecution of steps E7 and E8 of EXPANDPATH on the tree T_v.

5. Proving the performance guarantee of the R-join algorithm. To prove (2) of §3.1, we shall show that the cost of the feasible network produced by the algorithm is small relative to the number of cuts produced.

At any point in the execution of the algorithm, the *age* of a tree is the number of timesteps the tree grew. Thus the age of an active tree is the current number of elapsed timesteps, while the age of an inactive tree is the number of timesteps that had elapsed when the tree became inactive. We denote the age of a tree T by $age(T)$. We define the *connect-cost* of a call to the subroutine CONNECTTONETWORK as the number of edges added to the network by the routine not including any calls to the routine EXPANDPATH. That is, the cost for a call is the number of edges added in step C3, plus the cost of the recursive call in C9. We recursively define the *height* of a node to be 0 if it is a real node and one more than the maximum height of any node it encloses if it is a supernode.

LEMMA 5.1. *Steps* E7 *and* E8 *of* EXPANDPATH *are executed at most once for a given tree T through the course of the algorithm.*

Proof. Suppose we are about to begin the merging process for a given timestep. Through a series of calls to CONNECTTONETWORK, we build paths P that connect up some trees' networks. The key observation is that for every such path P, constructed in step C2 of CONNECTTONETWORK, every node of P except the last was previously free. (Recall that a *free* node is one that is not contained in any network N_T.) Moreover, since the edges of P are added to the network in step C3, such nodes are subsequently not free. Consequently, each node appears as a nonfinal node of a path P at most once during the course of the algorithm.

To complete the proof of the lemma, we need only add that a tree T for which steps E7 and E8 of EXPANDPATH(v, P) are executed corresponds to a nonfinal node v_i of the path P. □

LEMMA 5.2. *The connect-cost of a tree T is at most $age(T)$.*

Proof. We prove it by induction on the height of the nodes on the path from v to N_T. The statement trivially holds if all nodes have height 0, because by Proposition 4.1, v is at distance $age(T)$ from N_T.

Assume that the statement is true for nodes of height at most l. Let P be the path added in step C3, and let v_r be P's final node. By Proposition 4.1, P has at most $age(T)$ edges.

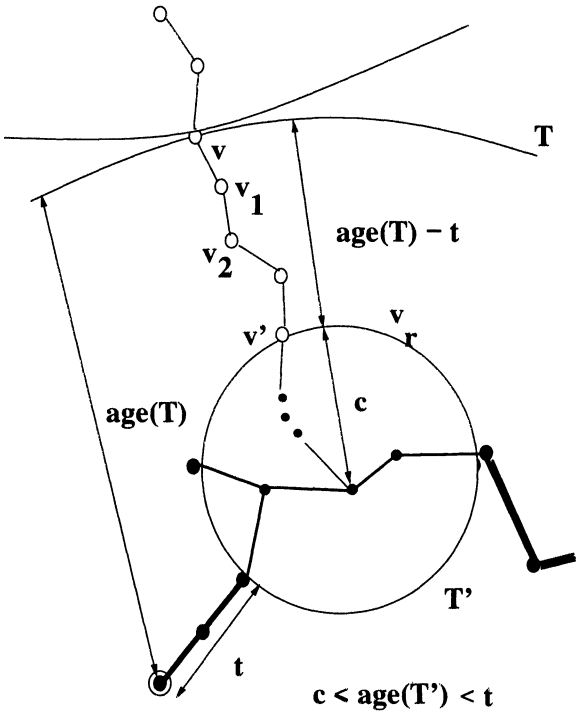


FIG. 4. *Proof of Lemma* 5.2 *that the cost of a call to* CONNECTTONETWORK *in the construction of the solution for the tree T is at most age(T).*

Therefore, if v_r is a real node, we are through. Otherwise, v_r corresponds to an inactive tree T'. The proof for this case is illustrated in Fig. 4. Let c be the cost of the recursive call CONNECTTONETWORK(v', T') in step C9. By the inductive hypothesis, c is at most $age(T')$, since no node in T' has height more than l. Suppose v_r was added to T after t timesteps. It follows that the number of edges in P is $age(T) - t$. Moreover, $age(T')$ is at most t, since T' was already inactive when v_r was added to T. Hence the total cost of the call to CONNECTTONETWORK which is $|P| + c$, is at most $age(T) - t + age(T')$, which in turn is at most $age(T)$. □

Define the *expand-cost* of a tree T as the cost of the calls CONNECTTO-NETWORK(v, T) and CONNECTTONETWORK(v', T) in steps E7 and E8. By Lemmas 5.1 and 5.2, the expand-cost of T is at most $2 \cdot age(T)$. Moreover, by the proof of Lemma 5.1, if the node v corresponding to T remains forever free, then these calls are never made, so the expand-cost of T is zero. We use $ExpandCost(T)$ to denote the expand-cost of T.

When trees $T_1, \ldots, T_r$ merge, the network N_T for the resulting tree T is constructed by taking the union of the networks for the T_i's, and then making some calls to CONNECTTONETWORK. We recursively define the cost of T as the sum of the costs of the trees merged to form T, plus the costs of the calls to CONNECTTONETWORK. Thus the cost of a tree T is the number of edges added to create N_T^*, not including edges added in steps E7 and E8 of EXPANDPATH. We denote the cost of T by $Cost(T)$.

We will charge the cost of a tree against the number of cuts assigned to the tree. Recall from the cut-packing algorithm that in each timestep we grow each tree, and assign the corresponding level cut to the tree. Moreover, when trees are merged, their cuts are assigned to the resulting tree. We denote the number of cuts assigned to a tree by $CP(T)$.

LEMMA 5.3. *After t timesteps have elapsed, the cost of a tree T is at most $2 \cdot CP(T) - 2 \cdot age(T)$.*

Proof. We shall prove this statement by induction on the number t of elapsed timesteps. When t is 0, the lemma holds trivially. Assume that the statement holds for t. During the $t+1$st timestep, each active tree T, is grown by one level, so $CP(T)$ goes up by one, while its age also increases by one. So far, so good. Next, trees are merged. The additional cost incurred in merging $T_1, \ldots, T_r$ to form a tree T is the cost of $2(r-1)$ calls to CONNECTTONETWORK, each at cost at most $age(T)$ by Lemma 5.2. Hence the total cost of T is

$$2(r-1)age(T) + \sum_{i=1}^{r} Cost(T_i)$$

which, by the inductive hypothesis, is at at most $2(CP(T) - age(T))$. □

Now we can bound the size of the feasible network output by our algorithm. The size is the sum, over all inactive trees T of the cost of T plus the expand-cost of T. For any tree T whose node remains free, the expand-cost is zero. Let us call a tree free if its corresponding supernode is free. Thus we have

$$\begin{aligned} &\text{size of feasible network} \\ &\leq \sum_{T} Cost(T) + ExpandCost(T) \\ &\leq \sum_{\text{free } T} Cost(T) + \sum_{\text{unfree } T} (Cost(T) + ExpandCost(T)) \\ &\leq 2(\sum_{T} CP(T) - \sum_{\text{free } T} age(T)), \end{aligned} \tag{5}$$

where the last inequality follows from Lemma 5.3 and our remarks about expand-cost.

Since $\sum_T CP(T)$ is the total number of cuts assigned by the cut-packing algorithm, we have proved a version of (2) with a factor of 2 instead of $2(1-1/k)$. To get the smaller factor, we prove a lower bound on the second sum in (5).

For a tree T, let k_T denote the number of sites that are nodes of T. Define $k_T^* = \sum\{k_{T'} : T \text{ encloses } T'\}$. Similarly, let $CP^*(T) = \sum\{CP(T') : T \text{ encloses } T'\}$.

LEMMA 5.4. *For any tree T, $age(T)$ is at least $CP^*(T)/k_T^*$.*

Proof. The key observation is that for any tree T', $CP(T')$ is at most $k_{T'}$ times $age(T')$, since each of the $k(T')$ sites is assigned a maximum of one cut per timestep until $age(T')$ timesteps. If T' is enclosed by T, then $age(T')$ is at most $age(T)$, so we have

$$\begin{aligned} CP^*(T) &= \sum\{CP(T') : T \text{ encloses } T'\} \\ &\leq \sum\{k_{T'} : T \text{ encloses } T'\} age(T) = k_T^* age(T). \end{aligned}$$ □

We use Lemma 5.4 to get our lower bound on $\sum\{age(T) : T \text{ free}\}$. Let $k^* = \max\{k_T^* : T \text{ free}\}$. Then by Lemma 5.4, for each free tree T, $age(T) \geq CP^*(T)/k^*$. Since each tree is enclosed by some free tree, $\sum\{CP^*(T) : T \text{ free}\}$ is the total number CP of cuts assigned. Hence

$$\sum\{age(T) : T \text{ free}\} \geq CP/k^*. \tag{6}$$

Substituting into (5) and replacing k^* by k, the total number of sites, gives (2) and completes the proof of Theorem 1.3.

6. Further directions for research. Two important variants of the basic Steiner tree problem in networks are the node Steiner problem [34], in which nodes are assigned costs, and the directed Steiner tree problem, in which the input graph is directed and one seeks a directed tree as the solution. It is an open problem to find good approximation algorithms for either problem. It was observed by Berman [5] that the set cover problem is reducible to the node Steiner problem via an approximation-preserving transformation. Khuller [20] made an analogous observation concerning the directed Steiner tree problem. In fact, Segev [34] gave an approximation-preserving reduction from the node Steiner problem to the directed Steiner problem. In view of Lund and Yannakakis' recent result showing that the set cover problem cannot be approximated by a factor smaller than logarithmic [28], it is natural to ask whether there are logarithmic-factor approximation algorithms for the node and directed Steiner problems. We have recently discovered such an algorithm for the former problem [24].

Acknowledgments. Thanks to Piotr Berman, Marshall Bern, Michel Goemans, Samir Khuller, Balaji Raghavachari, Arie Segev, and Richard Wong. Thanks also to the referees for a careful reading and helpful suggestions.

REFERENCES

[1] A. AGRAWAL, P. KLEIN, AND R. RAVI, *When trees collide: An approximation algorithm for the generalized Steiner tree problem on networks*, Technical Report CS-90-32, Brown University, Providence, RI, 1990.

[2] Y. P. ANEJA, *An integer linear programming approach to the Steiner problem in graphs*, Networks, 10 (1980), pp. 167–178.

[3] S. ARORA AND S. SAFRA, *Probabilistic checking of proofs: A new characterization of NP*, Proc. 33rd Symposium on Foundations of Computer Science, 1992, pp. 2–13.

[4] S. ARORA, C. LUND, R. MOTWANI, M. SUDAN, AND M. SZEGEDY, *Proof verification and hardness of approximation problems*, Proc. 33rd Symposium on Foundations of Computer Science, 1992, pp. 14–23.

[5] P. BERMAN, personal communication, 1991.

[6] P. BERMAN AND V. RAMAIYER, *Improved approximations for the Steiner tree problem*, Proc. 3rd Annual ACM-SIAM Symposium on Discrete Algorithms, 1992, pp. 325–334.

[7] R. T. CHIEN, *Synthesis of a communication net*, IBM J. Res. Develop., 3 (1960), pp. 311–320.

[8] C. J. COLBOURN, *The Combinatorics of Network Reliability*, Oxford University Press, London, 1987.

[9] ———, *Edge-packings of graphs and network reliability*, J. Discrete Math., 72 (1988), pp. 49–61.

[10] J. EDMONDS, AND E. L. JOHNSON, *Matching, Euler tours and the Chinese postman*, Math. Programming, 5 (1973), pp. 88–124.

[11] C. EL-ARBI, *Une heuristique pour le problem de l'arbre de Steiner*, RAIRO Rech. Oper., 12 (1978), pp. 207–212.

[12] U. FEIGE, S. GOLDWASSER, L. LOVÁSZ, S. SAFRA, AND M. SZEGEDY, *Approximating clique is almost NP-complete*, Proc. 32nd Symposium on Foundations of Computer Science, 1991, pp. 2–12.

[13] R. E. GOMORY AND T. C. HU, *Multi-terminal network flows*, J. Soc. Indust. Appl. Math., 9 (1961), pp. 551–570.

[14] M. X. GOEMANS, AND D. J. BERTSIMAS, *Survivable networks, linear programming relaxations and the parsimonious property*, Math. Programming, 60 (1993), pp. 145–166.

[15] H. N. GABOW, M. X. GOEMANS, AND D. P. Williamson, *An efficient approximation algorithm for the survivable network design problem*, Proc. 3rd MPS Conference on Integer Programming and Combinatorial Optimization, 1993, pp. 57–74.

[16] M. X. GOEMANS AND D. P. WILLIAMSON, *A general approximation technique for constrained forest problems*, Proc. 3rd Annual ACM-SIAM Symposium on Discrete Algorithms, 1992, pp. 307–315.

[17] F. K. HWANG AND D. S. RICHARDS, *Steiner tree problems*, Networks, 22 (1992), pp. 55–90.

[18] A. JAIN, *Probabilistic analysis of an LP relaxation bound for the Steiner problem in networks*, Networks, 19 (1989), pp. 793–801.

[19] R. M. KARP, *Reducibility among combinatorial problems*, in R. E. Miller and J. W. Thatcher, eds., Complexity of Computer Computations, Plenum Press, New York, 1972, pp. 85–103.

[20] S. KHULLER, personal communication, 1991.

[21] P. N. KLEIN, *A data structure for bicategories, with application to speeding up an approximation algorithm*, Inform. Process. Lett., 52 (1994), pp. 303–307.

[22] P. N. KLEIN, S. RAO, A. AGRAWAL, AND R. RAVI, *An approximate max-flow min-cut relation for multicommodity flow, with applications*, Combinatorica, to appear.

[23] M. X. GOEMANS, A. V. GOLDBERG, S. PLOTKIN, D. SHMOYS, É. TARDOS, AND D. P. WILLIAMSON, *Improved approximation algorithms for network design problems*, Proc. 5th ACM-SIAM Symposium on Discrete Algorithms, 1994, pp. 223–232.

[24] P. N. KLEIN AND R. RAVI, *A nearly best-possible approximation algorithm for node-weighted Steiner trees*, Proc. 3rd Symposium on Integer Programming and Combinatorial Optimization, 1993, pp. 323–332.

[25] L. KOU, G. MARKOWSKY, AND L. BERMAN, *A fast algorithm for Steiner trees*, Acta Inform., 15 (1981), pp. 141–145.

[26] J. KRARUP, *The generalized Steiner problem*, unpublished note, 1978.

[27] L. LÓVASZ, *An Algorithmic Theory of Numbers, Graphs and Convexity*, Society for Industrial and Applied Mathematics, Philadelphia, 1986.

[28] C. LUND AND M. YANNAKAKIS, *On the hardness of approximating minimization problems*, Proc. 25th ACM Symposium on Theory of Computing, 1993, pp. 286–293.

[29] K. MEHLHORN, *A faster approximation algorithm for the Steiner problem in graphs*, Inform. Proess. Lett., 27 (1988), pp. 125–128.

[30] M. MINOUX, *Network synthesis and optimum network design problems: Models, solution methods and applications*, Networks, 19 (1989), pp. 313–360.

[31] J. PLESNIK, *A bound for the Steiner tree problem in graphs*, Math. Slovaca, 31 (1981), pp. 155–163.

[32] R. RAVI AND P. N. KLEIN, *When cycles collapse: A general approximation technique for constrained 2-connectivity problems*, Proc. 3rd Symposium on Integer Programming and Combinatorial Optimization, 1993, pp. 39–55.

[33] V. J. RAYWARD-SMITH, *The computation of nearly minimal Steiner trees in graphs*, Internat. J. Math. Ed. Sci. Tech., 14 (1983), pp. 15–23.

[34] A. SEGEV, *The node-weighted Steiner tree problem*, Networks, 17 (1987), pp. 1–17.

[35] R. SRIDHAR AND R. CHANDRASEKARAN, *Integer solution to synthesis of communication network*, Technical Report, University of Texas, Dallas, TX, 1989.

[36] K. STEIGLITZ, P. WEINER, AND D. J. KLEITMAN, *The design of minimum-cost survivable networks*, IEEE Trans. Circuit Theory, CT-16, 4 (1969), pp. 455–460.

[37] G. F. SULLIVAN, *Approximation algorithms for Steiner tree problems*, Technical Report 249, Dept. of Comp. Sci., Yale University, New Haven, CT, 1982.

[38] H. TAKAHASHI AND A. MATSUYAMA, *An approximate solution for the Steiner problem in graphs*, Math. Japonica, 24 (1980), pp. 573–577.

[39] L. G. VALIANT, *The complexity of enumeration and reliability problems*, SIAM J. Comput., 8 (1979), pp. 410–421.

[40] D. P. WILLIAMSON, M. X. GOEMANS, M. MIHAIL, and V. V. VAZIRANI, *A primal-dual approximation algorithm for generalized Steiner network problems*, Proc. 25th ACM Symposium on Theory of Computing, 1993, pp. 708–717.

[41] P. WINTER, *Steiner problem in networks: A survey*, BIT 25 (1985), pp. 485–496.

[42] ———, *Generalized Steiner problem in outerplanar graphs*, Networks, 17 (1987), pp. 129–167.

[43] R. T. WONG, *Worst-case analysis of network design problem heuristics*, SIAM J. Alg. Discrete Meth., vol. 1 (1980), pp. 51–63.

[44] ———, *A dual ascent approach for Steiner tree problems on a directed graph*, Math. Programming, 28 (1984), pp. 271–287.

[45] A. Z. ZELIKOVSKY, *The 11/6-approximation algorithm for the Steiner problem on networks*, Algorithmica, 9, pp. 463–470.

SIAM J. COMPUT.
Vol. 24, No. 3, pp. 457–472, June 1995

003

RECTILINEAR PATH PROBLEMS AMONG RECTILINEAR OBSTACLES REVISITED*

CHUNG-DO YANG†, D. T. LEE‡, AND C. K. WONG§

Abstract. Efficient algorithms are presented for finding rectilinear collision-free paths between two given points among a set of rectilinear obstacles. The results improve the time complexity of previous results for finding the shortest rectilinear path the minimum-bend shortest rectilinear path, the shortest minimum-bend rectilinear path and the minimum-cost rectilinear path. For finding the shortest rectilinear path, a graph-theoretic approach is used and an algorithm is obtained with $O(m \log t + t \log^{3/2} t)$ running time, where t is the number of extreme edges of given obstacles and m is the number of obstacle edges. Based on this result an $O(N \log N + (m+N) \log t + (t+N) \log^2(t+N))$ running time algorithm for computing the L_1 minimum spanning tree of given N terminals among rectilinear obstacles is obtained. For finding the minimum-bend shortest path, the shortest minimum-bend rectilinear path, and the minimum-cost rectilinear path, we devise a new dynamic-searching approach and derive algorithms that run in $O(m \log^2 m)$ time using $O(m \log m)$ space or run in $O(m \log^{3/2} m)$ time and space.

Key words. rectilinear shortest path, minimum-bend path, path preserving graph, computational geometry, rectilinear obstacles

AMS subject classifications. 68U05, 68Q25, 68P05, 68R10

1. Introduction.

The problem of finding paths among obstacles has been extensively studied in the past. As the integrated circuits draw more research interest, finding rectilinear paths using different criteria has become an important variation of the traditional shortest path problem in automated circuit design. Both measures, the *number of bends* and the *length* of a rectilinear path, are two important factors while routing between two points among a set of rectilinear obstacles. Many efficient algorithms about finding rectilinear paths with respect to these two factors separately or jointly have been obtained [3], [4], [8], [10]–[14], [16], [18]. There are two particularly good results for finding shortest path (SP). The first, due to Clarkson, Kapoor, and Vaidya [3], runs in $O(m \log^{3/2} m)$ time, where m is the number of obstacle edges. The second, due to Wu et al. [16], runs in $O(m \log t + t^2 \log t)$ time, where t is the number of extreme edges of given obstacles. Mitchell [11] proposed a wave-front approach to find the shortest rectilinear path in $O(m \log^2 m)$ time, which is later reported to run in $\theta(m \log m)$ time after some modifications [12]. An edge on the boundary of an obstacle is an *extreme* edge if its two adjacent edges lie on the same side of the line containing the edge. Results from [3], [11], [12] are also applicable to any polygonal obstacles. Here we only focus on rectilinear paths. In this paper we shall present an $O(m \log t + t \log^{3/2} t)$-time algorithm that combines features of both results to find a shortest path. Finding the minimum spanning tree (MST) among N terminals can also be solved in $O(N \log N + (m+N) \log t + (t+N) \log^2(t+N))$ time based on the same method, which improves the $O(N \log N + (m+t^2) \log t)$-time algorithm due to Wu et al. [16].

With regard to the number of bends, Ohtsuki [13] proposed an $O(m \log^2 m)$-time and $O(m)$-space algorithm for finding a minimum-bend path (MBP). By combining their algorithms with the data structure from Imai and Asano [6], finding a minimum-bend path can be solved in $\theta(m \log m)$ time and space.

*Received by the editors April 20, 1992; accepted for publication (in revised form) October 27, 1993. This work was supported in part by the National Science Foundation under grants CCR-8901815 and CCR-9309743, and by the Office of Naval Research under grant N00014-93-1-0272.

†IBM Microelectronics, EDA Physical Design, E. Fishkill Facility, Hopewell Junction, New York 12533 (yangcd@fshvmfk1.vnet.ibm.com).

‡Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois 60208 (dtlee@eecs.nwu.edu).

§IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, New York 10598 (wongck@watson.ibm.com).

When both measures are considered jointly one may define problems by assigning different optimization priorities to length and bends. Recently, Yang, Lee, and Wong [17] presented algorithms to find a minimum-bend shortest path (MBSP), a shortest minimum-bend path (SMBP) and also a minimum-cost path (MCP) in $O(mt + m \log m)$ time, where t is the number of the extreme edges. We shall show below that this class of problems can be solved in $O(m \log^{3/2} m)$ time by applying a *hybrid* approach, called *dynamic-searching*, which is a combination of the graph-theoretic approach and the continuous-search (e.g., wave-front) approach. We first generate a graph just for guidance purposes and then by traversing the graph construct the corresponding rectilinear paths on the fly, thereby obtaining our goal path.

In the following, we first define our problems and give some preliminaries in §2. In §3, we show the faster algorithm for SP and MST. In §4, we focus on solving MBSP, SMBP, and MCP. We introduce the dynamic-searching approach and the guidance graph and then present our algorithms. The conclusion follows in the last section.

2. Preliminaries. A *rectilinear path* Π_{pq} is a path connecting two points p and q, which consists of only horizontal and vertical line segments.

Given two terminals s and d and a set of rectilinear obstacles, we define the following problems.

Problem SP is to find a path with the shortest distance. Such a path is called the shortest path, denoted *sp*.

Problem MBSP is to find a path with a minimum number of bends among all the shortest paths from s to d. Such a path is called the minimum-bend shortest path, denoted *mbsp*.

Problem SMBP is to find the shortest one among all the minimum-bend paths from s to d. Such a path is called the shortest minimum-bend path, denoted *smbp*.

Problem MCP is to find the minimum-cost path from s to d where the cost is a nondecreasing function f of the number of bends and the length of a path. Denote such a path *mcp*.

Each of the input obstacles is specified by a sequence of edges in clockwise order. An *extreme edge* is an obstacle edge with both of its adjacent edges lying on the same side of the line containing it. An *extreme point* is an endpoint of an extreme edge. Let m be the number of obstacle edges and t be the number of extreme edges. Let *EXTM* be the set of extreme points and let V be the set of obstacle vertices. Note that for a convex rectilinear polygon, there are only four extreme edges and eight extreme points. Much of the following discussion will be on a point set, $\{s, d\} \bigcup EXTM$, which will be denoted as *EXTMSD*.

We now define a basic graph generated by Clarkson, Kapoor, and Vaidya [3], which will be used in both of our algorithms for SP and for MBSP, SMBP and MCP.

DEFINITION 1. *Let R_{pq} denote the closed rectangle with segment $\overline{pq}$ as its diagonal. Given a set of points S, a point $q \in S$ is a* staircase point *of a point $p \in S$ if and only if R_{pq} does not contain any point in S other than p and q. Let $SC(p)$ denote the set of staircase points of point p.*

We recall an important property employed by Clarkson, Kapoor, and Vaidya [3] and refer to their algorithm as Algorithm C in what follows. That is, a shortest path from one point p to some other point q can always be replaced by subpaths going from p to one of its staircase points and the same holds for subsequent subpaths. With this property, Algorithm C obtains a graph on the vertex set, $V \bigcup \{s, d\}$, by adding all the edges between points and their staircase points. Such a graph, which preserves shortest paths, may contain $O(m^2)$ edges and $O(m)$ vertices, which yields a basic $O(m^2)$-time path-finding algorithm. Then by introducing *Steiner points* to the graph, they reduce the graph to have $O(m \log m)$ vertices and edges, referred to as Algorithm C1. Algorithm C1 constructs the graph in a recursive manner. A

cutting line that divides the vertices in two halves is introduced first. Then all the vertices are projected onto this line, creating *Steiner points*. Edges are added between vertices and their corresponding Steiner points, and between consecutive Steiner points. Recursively perform the cutting, projecting and adding edges on the subsets of vertices that lie on both sides of the cutting line. Since each vertex can create at most $O(\log m)$ Steiner points, the number of vertices of the graph is $O(m \log m)$. The edges, which are either horizontal or vertical, are also added during the creation of Steiner points. Therefore, it is not difficult to see that there are $O(m \log m)$ edges. The time complexity to search this graph for a shortest path is thus $O(m \log^2 m)$.

To further improve the time complexity, they use a *grouping* technique to generate more edges but fewer vertices, i.e., $O(m \log^{3/2} m)$ edges and $O(m \log^{1/2} m)$ vertices, for the graph and are able to reduce the time complexity of the searching to $O(m \log^{3/2} m)$. This algorithm is referred to as Algorithm C2. The grouping technique is to virtually cut the space into strips before they generate Steiner points such that each strip contains only $O(\sqrt{\log m})$ points. In each strip, only two Steiner points are created (the highest and the lowest) to provide connections across strips. Inside a strip, an edge is created for each pair of vertices that lie on different sides of the cutting line. It can be proved that such a graph still preserves the shortest path. Note that this refined graph contains some *oblique* edges due to the direct connections inside each strip. This property makes this graph different from the previous one when we use it in our algorithm later. We refer the reader to [3] or [10] for details of these algorithms.

By applying Fredman and Tarjan's $O(|E| + |V| \log |V|)$ shortest path algorithm [5] on the final graph, they obtain the shortest path in $O(m \log^{3/2} m)$ time.

On the other hand, the algorithm of Wu et al. [16], referred to as Algorithm W, finds the rectilinear shortest path among rectilinear obstacles by first constructing a so-called "*track graph*" based on the extreme edges. Horizontal and vertical tracks, which are projected from extreme edges, are the edges and the intersections of these tracks are the vertices of the track graph. The graph plus the graph representing the obstacle vertices and edges, is of size $O(m + t^2)$, hence yielding an $O(m \log t + t^2 \log t)$ time algorithm.

Depending on the values of t and m, Algorithm W may be asymptotically more efficient than Algorithm C2 and vice versa.

Before we present our algorithm, which is better than both, let us give some preliminary results.

DEFINITION 2. *A* U-shaped *subpath* (*or* U-subpath *for short*) *consists of three segments, s_1, s_2, and s_3 such that s_1 and s_3 lie on the same side of the line containing s_2; segment s_2 is referred to as the* U-segment *of the* U-*subpath. A* staircase *path is a path containing no* U-*subpath.*

LEMMA 2.1 (see Fig. 1). *Any sp, mbsp, smbp, or mcp, $\Pi_{s,d}$, from s to d can be divided into a sequence of subpaths $\pi_i, i = 1, \ldots, m$, such that all π_i's are staircase paths and the start and endpoints of all π_i are points in EXTMSD.*

Proof. If the U-segments of all the U-subpaths of $\Pi_{s,d}$ contain extreme points, then each U-subpath can be cut into subpaths at these extreme points and each subpath will satisfy the property as claimed. Consider now a U-subpath of $\Pi_{s,d}$ denoted as s_1, s_2, and s_3, such that its U-segment does not contain any extreme point (as in Fig. 1). We can always *shorten* the U-subpath by shrinking s_1 and s_3 and moving s_2 accordingly without incurring any bends. A strictly better *sp*, *smbp*, *mbsp*, or *mcp* can therefore be obtained, which yields a contradiction. That is, any *sp*, *mbsp*, *smbp*, or *mcp* can be represented by a sequence of staircase subpaths starting and ending at points in *EXTMSD*. □

The above lemma implies that it is sufficient to just consider staircase paths while deriving efficient algorithms.

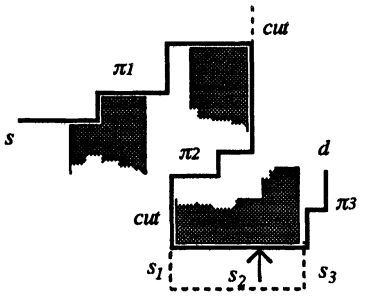


FIG. 1. *Divide a path into staircase subpaths.*

3. A faster algorithm for SP.

3.1. A smaller path-preserving graph. Our algorithm is based on the following observation that it is sufficient to generate a shortest-path-preserving graph by just considering the extreme points and their projection points when using Algorithm C1 or C2.

DEFINITION 3. *Given a set of obstacles, define* the projection point set $\mathcal{P}(p)$ *of a point* p *to be* $\{q | q$ *is on some obstacle boundary and* $\overline{qp}$ *is a horizontal or vertical collision-free segment*$\}$.

Let $PJ = \{q | q \in \mathcal{P}(p), p \in EXTMSD\}$ and $I = EXTMSD \bigcup PJ$ denote the set of *essential points*, which are used in the construction of the backbone of the shortest-path-preserving graph. They are not, however, the only vertices in the final graph.

DEFINITION 4. *Define the graph* $SPG0$ *to be a graph with vertex set equal to* I *and edge set equal to* $\{(a, b) | a, b \in I, a \in SC(b)\}$. *The cost of edge* (a, b) *is the rectilinear distance between points* a *and* b.

The graph $SPG0$ is the backbone of the graphs that we generate later. $SPG0$ contains in the worst case $O(|I|)$ (or $O(t)$) vertices and $O(|I|^2)$ (or $O(t^2)$) edges. We adopt the graph reduction method used in Algorithm C1 by introducing Steiner points. Let SPG_r denote the reduced graph we generate. Let SPG_{rr} denote the refined, reduced graph when the method of Algorithm C2 is used. We have the following lemma whose proof is similar to that given in [3], [10], [14].

LEMMA 3.1. SPG_r *has* $O(|I| \log |I|)$ *vertices and edges, and can be computed in* $O(m \log |I| + |I| \log |I|)$ *time.* SPG_{rr} *has* $O(|I| \log^{1/2} |I|)$ *vertices and* $O(|I| \log^{3/2} |I|)$ *edges, and can be computed in* $O(m \log |I| + |I| \log^{3/2} |I|)$ *time.*

DEFINITION 5. *Define the boundary graph* BG *to be a graph with vertex set equal to* $EXTM \bigcup PJ$ *and edge set equal to the set of edges connecting* p *and* q *in the vertex set that lie consecutively on the boundary of an obstacle. The cost of the edge* (p, q) *is defined to be the shorter rectilinear path length along the boundary of the obstacle connecting points* p *and* q.

Let SPG be the union of SPG_{rr} and BG. The following lemma is easily established.

LEMMA 3.2. SPG *contains* $O(t \log^{1/2} t)$ *vertices and* $O(t \log^{3/2} t)$ *edges.*

We now prove that SPG is a shortest-path-preserving graph. Since the extreme points are in SPG, from Lemma 2.1 it is sufficient to show that for any shortest path all its subpaths connecting points in $EXTMSD$ are embedded in SPG.

LEMMA 3.3. *If there exists a shortest staircase path* RP *connecting two points* $p, q \in EXTMSD$, *then there is a path* P *in* SPG *connecting* p *and* q *with the same length as* RP.

Proof. Denote the abscissa and ordinate of a point p as $p.x$ and $p.y$, respectively. Without loss of generality, let $p.x \leq q.x$ and $p.y \leq q.y$, as shown in Fig. 2(a). Let the sequence of adjacent segments in RP be denoted as $h_1, v_1, h_2, v_2, \ldots, h_g, v_g$, where h_i's are horizontal

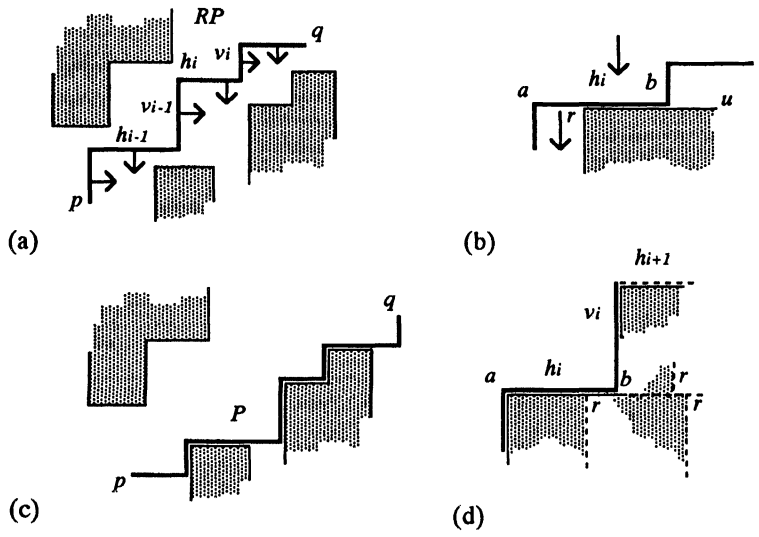


FIG. 2. *Situations while dragging paths.*

segments and v_i's are vertical segments. h_1 and v_g can be empty, respectively. Apparently, the path is in R_{pq}. If there is no obstacle intersecting R_{pq}, then p and q are staircase points of each other, and P is an edge in SPG. Suppose that there are obstacles in R_{pq}. We perform the following dragging operations to h_i from $i = 1$ to $i = g$:

(1) Drag h_i downward until it either is as low as p, hits some obstacle, or is as low as h_{i-1}. In the last case, merge h_i and h_{i-1} to be one horizontal edge.

(2) If $h_i = (a, b)$, $a.x < b.x$, hits some obstacle on edge (r, u) but $r \neq a$, then break h_i into two segments, (a, r) and (r, b) and perform dragging operations on (a, r) (Fig. 2(b)).

(3) At each dragging, the adjacent vertical segments are adjusted accordingly.

We then perform the same dragging operations to the vertical segments on RP except that we drag them rightward and adjust horizontal segments accordingly.

Denote as P the resultant path (Fig. 2(c)) after performing these operations. It is not hard to see that P has the following properties: first, it has the same length as RP; second, it is a staircase path; third, all the turning points between h_i and v_{i-1} (after re-ordering segments on P as $\{h_1, v_1, \ldots, h_{g'}, v_{g'}\}$) will be an *upper-left* corner of an obstacle. Since we focus only on paths between points in *EXTMSD*, we may assume that there is no extreme point on P. Otherwise, those subpaths formed by cutting P at those extreme points can be considered respectively.

Consider a horizontal edge $h_i = (a, b)$, $1 < i \leq m'$ (treat all the other horizontal segments similarly and all vertical ones symmetrically). As in Fig. 2(d), point a is either p or a corner vertex of an obstacle as shown before. Let (a, r) be the edge of obstacle o containing a.

(1) If the boundary of o turns downward at r and $r.x \leq b.x$, then r is an extreme point on (a, b), which violates our assumption.

(2) If it turns upward at r, $r = b$ is a vertex of o.

(3) If it turns upward or downward at r where $r.x > b.x$ and h_{i+1} exists, then v_i must align with an *extreme* edge γ (due to the rightward dragging we performed on v_i) of a possibly different obstacle and b is its projection on o. If h_{i+1} does not exist in this case, then v_i must connect q, which also makes b a projection point. Thus, b is a projection point on o. Therefore h_i is part of an edge in the boundary graph.

All edges in P satisfying (2)–(3) are either connecting extreme points to the projections or connecting points in $EXTM \bigcup PJ$ along the boundary of obstacles. The first kind of edges are all in SPG_{rr}, and the second kind of edges are in BG and thus also included in SPG. Note that the dragging operations may be performed symmetrically by shifting horizontal segments upwards and vertical segments leftward. Since all the edges or subpaths of P are embedded by SPG, the lemma is proved. □

We conclude with the following theorem.

THEOREM 3.4. *SPG is a shortest-path-preserving graph.*

Now we summarize the algorithm to construct SPG and find the shortest rectilinear path.

Algorithm FindSRP.

1. Sort all the obstacle edges on their X and Y coordinates.
2. By plane-sweeping horizontally and vertically, find the projection set PJ. Projection points are recorded in order in lists associated with every obstacle edge. Each such list can be implemented as a balanced binary search tree [1].
3. Find the boundary graph BG along all obstacle boundaries by using $EXTM$ and PJ as its vertex set. This is done by traversing the obstacle boundary and linking up all the extreme points and the projection points in lists associated with obstacle edges.
4. Find SPG_{rr} as in Algorithm C2 using $EXTMSD \bigcup PJ$ as the vertex set.
5. Merge SPG_{rr} and BG to obtain SPG.
6. Find the shortest path from s to d on the graph SPG using Fredman and Tarjan's algorithm [5].

THEOREM 3.5. *The Algorithm FindSRP finds the shortest rectilinear path from s to d among rectilinear obstacles in $O(m \log t + t \log^{3/2} t)$ time using $O(m + t \log^{3/2} t)$ space.*

Proof. Steps 1 and 2 take $O(m \log t)$ time. For step 3, computing BG can be done in $O(m)$ time by traversing along the obstacle boundaries. For step 4, construction of SPG_{rr} can be computed in $O(m \log t + t \log^{3/2} t)$ time. The reader is referred to [3], [10], [14]. Since SPG_{rr} contains $O(t \log^{3/2} t)$ edges and $O(t \log^{1/2} t)$ vertices, and BG contains $O(t)$ edges and vertices, the combined SPG can be computed in time linear in the size of SPG. Finally, in step 6, applying Fredman and Tarjan's shortest path algorithm on SPG that runs in $O(|E|+|V| \log |V|)$ time, we are able to find the shortest rectilinear path in $O(t \log^{3/2} t)$ time. Overall, the time needed is of $O(m \log t + t \log^{3/2} t)$. The space needed is $O(m + t \log^{3/2} t)$. The correctness of the algorithm follows immediately. □

3.2. Finding MST among obstacles. With the same approach, one can handle multiple input points and generate a similar graph where the minimum spanning tree (MST) is preserved. Given N terminals, Wu et al. [16] obtained the MST in $O(N \log N + (m+t^2) \log t)$ time based on the *track graph*. Here we simply let the essential point set be the same as we used before with all the N input terminals included. As mentioned earlier, based on Algorithm C1, we can generate SPG_r containing only $O(|I| \log |I|)$ edges and vertices for the essential point set I. Combining this graph with the boundary graph we have a different shortest-path-preserving graph, denoted SPG_m, with $O((t + N) \log(t + N))$ edges and vertices.

Since the pairwise shortest paths are all retained in SPG_m, the MST is also retained. We then adopt the algorithm presented by Wu et al. [15], [16] that runs in $O(|E_G| \log |V_G|)$ time for finding an MST among a set of points S on a graph $G = (V_G, E_G)$ with $S \subseteq V_G$.

THEOREM 3.6. *The minimum spanning tree of N terminals among obstacles can be computed in $O(N \log N + (m + N) \log t + (t + N) \log^2(t + N))$ time, where m is the number of obstacle edges and t is the number of the extreme edges of obstacles.*

4. Faster algorithms for MBSP, SMBP, and MCP. We now deal with problems, MBSP, SMBP, and MCP. We shall refer to any of *mbsp*, *smbp*, or *mcp* as an *optimal path*.

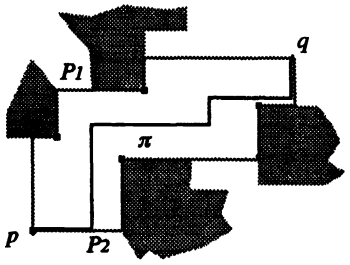


FIG. 3. *The corridor of a path.*

DEFINITION 6. *A point p is said to* 1-dominate *a point q if the X- and Y-coordinates of p and q satisfy the condition* $p.x \geq q.x$ *and* $p.y \geq q.y$. *In other words, point p lies in the first quadrant when point q is placed at the origin.* 2,3,4-dominating *relations are defined symmetrically.*

Redefining the *essential point set* in Definition 3.1 to be $V \bigcup \{s, d\}$, and using Algorithm C1, we obtain a graph, referred to as *guidance graph*, and denoted as GG. The guidance graph will be used to *guide* our path finding algorithm. The graph was also exploited by Lee, Yang, and Chen [10] to solve shortest path problems when the obstacles are *weighted.* We summarize some properties of GG as follows.

LEMMA 4.1 ([3], [10], [14]). *GG has the following properties:*

(1) It has $O(m \log m)$ edges and vertices.

(2) V is a subset of the vertex set of GG.

(3) All edges in GG are horizontal or vertical.

(4) For any two points p and q in V, if R_{pq} is empty, then there is a shortest path between p and q in GG.

(5) It can be constructed in $O(m \log m)$ time using $O(m \log m)$ space.

We show below that the guidance graph contains at least a path that is homotopic to an optimal path.[1] Lemma 2.1 implies that if for any staircase subpath π of an optimal path between two points p and q in V, we can find a path in GG between p and q that is homotopic to π, then there is always a path in GG that is homotopic to the optimal path. We introduce the *corridor* of a staircase path between two points p and q in V. We consider only the staircase (first type) where point q 1-dominates point p. The corridor of a staircase (second type) where q 2-dominates p is defined similarly. Without loss of generality, a staircase in the following discussions refers to the first type. The second type of staircase can be treated similarly. Let π_{pq} denote the staircase path in R_{pq} for $p, q \in V$ and q 1-dominates p. Let $D(\pi_{pq})$ and $U(\pi_{pq})$ denote the sets of vertices in V that lie in R_{pq} and below and above π_{pq}, respectively.

DEFINITION 7 (see Fig. 3). *Given two points* $p, q \in V$, *with q* 1*-dominating p, and a staircase path* π *from p to q, define* corridor(p, q, π) *to be the* open *rectilinear region enclosed by two staircase paths from p to q, denoted* P_1 *and* P_2, *where* P_1 *and* P_2 *satisfy the following:* P_1 *passes through and makes upward turns at points in* $U(\pi_{pq})$ *that are not* 4*-dominated by any other point in* $U(\pi_{pq})$ *and* P_2 *passes through and makes rightward turns at points in* $D(\pi_{pq})$ *that are not* 2*-dominated by any other point in* $D(\pi_{pq})$.

One can see that the *corridor*(p, q, π) does not contain any points of V and all staircase paths between p and q in *corridor*(p, q, π) are homotopic to each other.

LEMMA 4.2. *There exists a path* π *from s to d in GG such that* π *is homotopic to an optimal path from s to d.*

[1] Two paths are homotopic to each other if one can be continuously dragged to become the other without crossing any points in V.

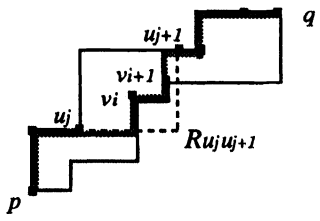


FIG. 4. *A path in the corridor and GG.*

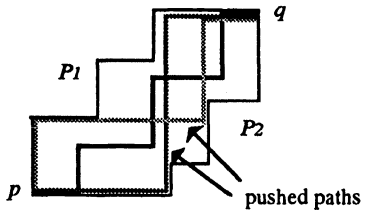


FIG. 5. *Two pushed paths from p in a corridor.*

Proof. From Lemma 2.1, we can focus on a staircase subpath between two points p and q in V. Let π^*_{pq} denote a staircase subpath from p to q of an optimal path. Let $u_{i,i=1,\ldots,k}$ and $v_{i,i=1,\ldots,t}$ represent the points in V on P_1 and P_2 that define *corridor*(p, q, π^*_{pq}), respectively (Fig. 4). v_i 1-dominates v_j if $i > j$ and u_k 1-dominates u_l if $k > l$. Now let $z_{i,i=1,\ldots,r}$ be any *maximal* sequence such that z_i is in $u_{i,i=1,\ldots,k}$ or in $v_{i,i=1,\ldots,t}$, and z_i 1-dominates z_j if $i > j$. Apparently, all z_i's are in V and for every two consecutive elements z_i and z_{i+1}, there is no other point in $R_{z_i z_{i+1}}$. According to the definition of GG, GG provides the connection between z_i and z_{i+1} and thus there is a path in GG connecting p and q through $z_{i,i=1,\ldots,r}$. We therefore conclude that there is a path from s to d embedded in GG that is homotopic to any optimal path from s to d. □

DEFINITION 8 (see Fig. 5). *Define a pushed staircase path from p to q in corridor*(p, q, π) *to be a staircase path from p to q, such that the path only makes turns alternately at points on* P_1 *and* P_2. *Any two-segment subpath, connecting a horizontal and a vertical segment, is called an* L-subpath. *A* canonical path *from s to d is a concatenation of pushed staircase paths connecting two points p and q with the last segment of each staircase path overlapping the leading segment of the next pushed staircase path.*

LEMMA 4.3. *There are at most four different kinds of pushed staircase paths from p to q in the corridor between p and q for any* $p, q \in V$.

Proof. (see Fig. 5) Starting from either p or q, there are at most two pushed staircase paths, one starting horizontally and the other starting vertically. Once we decide on the first segment of the pushed path from either p or q, the rest of it is unique. □

LEMMA 4.4. *There exists an optimal path from s to d such that any of its staircase subpaths between two consecutive points* $p, q \in EXTMSD$ *is a pushed path.*

Proof. Consider a subpath between p and q of an assumed optimal path. Let it be Π^*_{pq}. Without loss of generality, let q 1-dominate p. Consider *corridor*(p, q, Π^*_{pq}) as a polygon containing Π^*_{pq}. Since there are no points in V inside the corridor, we can, starting from p, drag all the horizontal segments of Π^*_{pq} upward and vertical segments rightward until they hit the boundary of the corridor without increasing the length or the number of bends. Consequently we obtain a pushed path from p to q with the same length and number of bends as Π^*_{pq}. This applies to all such subpaths in an optimal path. The lemma is proved. □

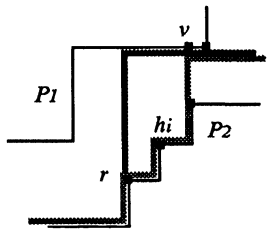


FIG. 6. *An L-subpath.*

This lemma implies that there exists an optimal path which is a canonical path.

LEMMA 4.5. *For a pushed staircase path* Π *from* p *to* q, *there is a path* P_{GG} *in* GG *from* p *to* q *passing through all the essential points on* Π.

Proof. According to the definition of pushed path, Π is a sequence of horizontal or vertical segments or L-subpaths connecting points in V. Since every horizontal or vertical segment between two points, u and v in V on path Π, is obviously an edge in GG, we can focus on proving that each L-subpath is also supported in GG.

Without loss of generality (Fig. 6), consider an L-subpath that goes from $r \in D(\Pi_{pq})$ upward and then turns rightward to $v \in U(\Pi_{pq})$ without containing any other points in V. If R_{rv} is empty, then we are sure that the connection between r and v is supported in GG. Otherwise, there are some essential points on P_2 falling in R_{rv}. Order them as $h_i, \{i = 1, \ldots, k\}$ such that R_{r,h_1}, $R_{h_i,h_{i+1}}$ and $R_{h_k,v}$ are all empty. Based on the properties of GG, between every two consecutive points of $\{r, h_1, h_2, \ldots, h_k, w\}$ there exists a path in GG. Therefore, the L-subpath between r and v is also supported. This completes the proof. □

We call the path P_{GG} described in Lemma 4.5 a *target path*. We intend to search on GG a target path and convert it to a canonical path by some *dragging* operations. Since GG contains only vertical or horizontal edges, we always append a vertical or horizontal segment to the end of the computed pushed path when we advance in the graph searching process. The last segment of a pushed path may be subject to dragging operation. When we reach d, we have a pushed path, which will be an optimal path. By focusing only on dragging the target path, we define the *dragging* operations applied to segments of a path.

DEFINITION 9. *Let* $last(\pi)$ *be the last staircase subpath of* π *from* p *to* v, *where* p, $v \in V_{GG}$. *A segment* w *of* $last(\pi)$ *is fixed if either*

(1) w *is horizontal (respectively, vertical) and cannot be dragged any further vertically (respectively, horizontally) without crossing any point in* V, *or*

(2) *it is the first segment of* $last(\pi)$.

It is floating *otherwise.*

When we advance along a target path on GG, the fixed portion of the path remains fixed thereafter and need not be considered again. Only the floating segment of the path needs to be considered for possible dragging, as defined below.

DEFINITION 10. *Let* π_{sp} *be a pushed path from* s *to* p, *and let* π'_{sq} *be the path formed by concatenating to* π_{sp} *an edge* $e \in E_{GG}$ *from* p *to* q, *where* e *is either horizontal or vertical (Fig. 7). Let* π_{sq} *be the pushed path dragged from* π'_{sq} *by the following dragging operations. Let* w *and* w_{new} *be the last segments of* π_{sp} *and* π_{sq}, *respectively, and assume they are horizontal. The vertical case can be defined similarly. Define the dragging operations on* π'_{sq} *as follows*:

(1) q *is to the right of* p (e *is horizontal*) (*Fig.* 7(a)): $w_{new} = w||e$ *is of the same type (fixed or floating) as* w, *where "||" denotes path concatenation. Note that* w *is fixed if it borders an obstacle above.*

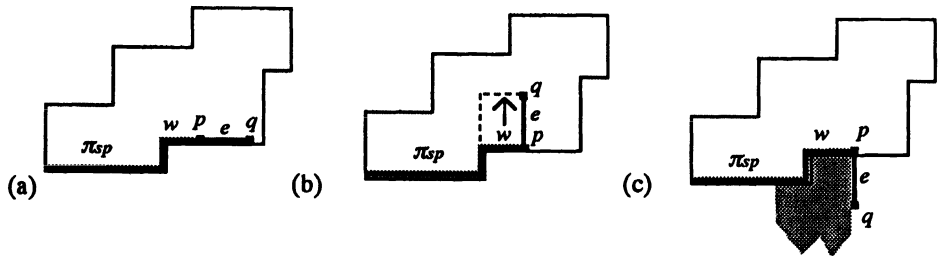


FIG. 7. *Dragging operations while advancing on e.*

(2) *q is to the left of p: Since w is horizontal, this is not possible.*

(3) *q is above p (e is vertical):*

(3.1) *w is fixed: Let w_{new} be e. w_{new} is fixed if it borders an obstacle to the right.*

(3.2) *w is floating: (Fig. 7(b)) Drag w upward and adjust π accordingly:*

(3.2.1) *If w can be dragged to q without hitting a point in V, then w_{new} is the dragged w.*

(3.2.2) *Otherwise, we ignore this advancing step.*

(4) *q is below p (e is vertical): (Fig. 7(c))*

(4.1) *If w borders an obstacle below it, we have a U-subpath here. We now have a new staircase subpath (of the second type) with w being the leading segment, which is fixed, and last$(\pi_{sq}) = w||e$.*

(4.2) *Otherwise, we ignore it.*

LEMMA 4.6. *A target path in GG can be successfully transformed by the dragging operations defined above to be a canonical path which is an optimal path.*

Proof. We can follow Lemma 4.5 and just focus on an L-subpath of an optimal path between two points in V. If each L-subpath from u to v can be formed by dragging a corresponding subpath on the target path from u to v, then the lemma is proved. According to Lemma 4.5 the corresponding subpath on the target path consists of horizontal and vertical edges which form a staircase path from u to v. There is no point of V in the area between this subpath and our L-shaped optimal subpath from u to v. Clearly, we will not hit any point in V when we do the dragging. Operations (1) and (3) are sufficient to drag edges in such a staircase path to form the L-subpath. Operation (3.2.2) ignores the case when we hit some point during dragging. Step (4.1) is just for making turns around a boundary edge and forming a U-subpath. Step (4.2) ignores the cases not belonging to the target path. □

4.1. The Algorithm and Its Complexity. Now we are ready to describe our algorithm: we apply Dijkstra's shortest path searching algorithm [1] on the graph GG to find the optimal path. While searching on GG and computing the pushed path, more than one pushed path may be generated when we reach a vertex from different edges. The information of the pushed paths leading to a vertex will be computed, and compared. Only the *best path(s)* obtained thus far will be retained at each vertex. We discuss below only the algorithm for MBSP. Modifications to the algorithms for solving SMBP or MCP are straightforward.

Algorithm MBSP.

1. Construct the guidance graph GG, and preprocess V for the dragging operations. This can be done using method in [2], which supports $O(\log m)$ query time using $O(m \log m)$ preprocessing time and linear space.
2. Apply Dijkstra's algorithm to find the optimal path according to metric vector $v = (d(\pi), b(\pi))$, where $d(\pi)$ and $b(\pi)$ denote the length and the number of bends of π, respectively. Let π_u be a pushed path obtained so far from s to a vertex u on GG.

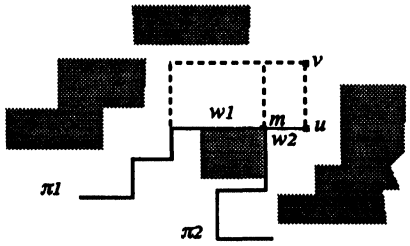


FIG. 8. *One path subsumes the other.*

There may be other pushed paths computed while reaching u from other edges and have been stored in u. We thus store at u the *type* of staircase of the last staircase subpath. Do the following:

2.1. If π_u is strictly worse than any other stored in u, discard π_u.

2.2. If π_u is strictly better (with lexicographically smaller metric) than some of the pushed paths stored in u, we discard the worse ones and store π_u at u.

2.3. If two have the same metric and the same type, compare their last segments. If their last segments overlap, then discard the one with the longer last segment. Otherwise (If they are not of the same type), keep both of them in u.

2.4. Calculate the metric vector of a pushed path advancing to each neighbor v of u via edge (u, v) in GG and put that into the queue used by the Dijkstra's algorithm. Note that there are at most eight pushed paths stored at each vertex u. That is, for each type of path there are two pushed paths, with the last segment leading to u either horizontally or vertically.

When we reach a vertex u, we ignore a pushed path if there exists one with smaller metric. When two pushed paths have the same metric, yet reaching u in different directions, we simply keep them all in u and proceed with the next advancing. We are able to compare and ignore some of those pushed paths that are of the same type and reach u in the same direction. See Lemma 4.7 below. The number of pushed paths that need to be stored in each vertex u is therefore at most eight, i.e., four pushed paths from vertex v that i-dominates u, $i = 1, 2, 3, 4$, and the last segment to u can be either vertical or horizontal.

LEMMA 4.7. *Consider two pushed staircase paths π_1 and π_2 with the same metric vector from s to some vertex u in GG. If the last segment, w_1, of π_1 is longer than the last segment, w_2, of π_2 and w_1 contains w_2 (Fig. 8), then any pushed path generated using π_1 will not be better than the pushed path generated using π_2. The same holds for other types of paths.*

Proof. Without loss of generality, let w_1 and w_2 be horizontal. Let the other endpoint of w_2 be m (u is the other endpoint). If w_1 is fixed, then obviously π_2 can replace π_1 in any further case. If w_1 is floating, then we may obtain a pushed path from π_1 by dragging w_1 upward. Wherever w_1 goes, the area swept over by it is empty. Since w_2 is shorter, it can also be dragged to the same position as w_1 with the same vertical distance offset. The dragging adds the same vertical distance offset to both paths and yet incurs no extra bend to either path. That is, any pushed path π_1 can grow to is no better than the best pushed path grown from π_2. The lemma is proved. □

THEOREM 4.8. *The problems MBSP, SMBP, and MCP can be solved in $O(m \log^2 m)$ time, using $O(m \log m)$ space.*

Proof. According to Lemma 4.6, while traversing the target path and applying the dragging operations on the way, we can get an optimal path from s to d. Lemma 4.7 guarantees that when a path is discarded, there must be some other path that subsumes it. This ensures that the final path is an optimal path. As to the running time, according to Lemma 4.1, the time

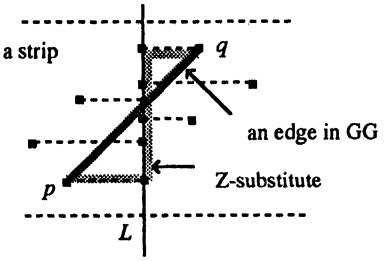


FIG. 9. *The Z-substitute.*

needed to generate the graph GG is $O(m \log m)$. The time used for searching on GG from s to d follows the complexity of Dijkstra's algorithm which is $O(|E| \log |E|)$ on a graph $G(V, E)$. The time for searching on GG is therefore $O(m \log^2 m)$. When we advance on GG through an edge, we perform dragging operations on at most eight pushed paths. Since each dragging operation can be done in $O(\log m)$ time [2] with $O(m \log m)$ time and $O(m)$ space preprocessing, the time complexity of the algorithm is $O(m \log^2 m)$. We need space for storing the guidance graph ($O(m \log m)$) and the pushed paths. Each pushed path has size less than that of its corresponding target path on GG and an edge of GG can be traversed (appended and dragged) by at most eight pushed paths from one of its end points. Therefore, the size of all the pushed paths kept while searching will be $O(E_{GG})$ or $O(m \log m)$. □

4.2. Solving MBSP, SMBP and MCP in $O(m \log^{3/2} m)$ time. The algorithm can be modified to run in $O(m \log^{3/2} m)$ time as follows. First, we find a more suitable guidance graph and a more efficient searching algorithm. Algorithm C2, as mentioned, can construct a refined, reduced graph, SPG_{rr}, providing the same connections as GG, with $O(m \log^{3/2} m)$ edges and $O(m \log^{1/2} m)$ vertices in $O(m \log^{3/2} m)$ time. We adopt this graph as the new *guidance graph*, denoted GG', and use the shortest path searching algorithm of Fredman and Tarjan [5], which runs in time $O(|E| + |V| \log |V|)$. The algorithm thus spends only $O(m \log^{3/2} m)$ time on graph searching. However, one of the properties in Lemma 4.1 does not hold any more, that is, the edges in GG' are not necessarily horizontal and vertical. This has effects on our dragging operations since we can only deal with horizontal and vertical segments in our dragging. We thus do some adjustments to the graph and compute substitutes for all such oblique edges.

From [3], each oblique edge in GG' is constructed from a set of horizontal and vertical edges in GG, which forms a *Z-shaped subpath* consisting of three segments where the first and the third lie on the different sides of the line containing the second one (Fig. 9). We call these three segments the *Z-substitute* of the original oblique edge. We can perform dragging similarly on these three segments when we encounter an oblique edge while searching on the new guidance graph and transform the target path to a pushed path.

To handle oblique edges, we add one dragging operation to the previous definition.

DEFINITION 11. *Define dragging operations as follows (assume last(π) is a staircase path):*

(1)–(4) *The same as operations* (1)–(4) *in Definition* 10.

(5) *For an oblique edge, replace it by the three segments of the corresponding Z-substitute. Apply the same operations as* (1)–(4) *to each of them in sequence.*

Besides reducing searching time on the guidance graph GG', we must devise a better way to perform dragging, since spending $O(\log m)$ time for each dragging consumes overall $O(|E_{GG'}| \log m)$ time, which becomes $O(m \log^{5/2} m)$.

In the following, we compute the *hit vertices* for each edge (at most one in each direction) when we generate the the graph GG' in $O(m \log^{3/2} m)$ time. Thereafter, while searching and computing the path, we are able to obtain in $O(1)$ time the hit vertex for each dragging operation, thus realizing a total of $O(m \log^{3/2} m)$ time. Note that the dragging operations are applied to the last segment of a pushed path (i.e., the w_{new} in the dragging definition) which may consist of several edges of the graph GG'. For example, consider that we form w_{new} by concatenating a horizontal last segment w with a horizontal edge e on GG. We simply let the hit vertex of w_{new} be the lower one of the hit vertices of w and e when we drag w_{new} upward. Dragging horizontal segments downward or dragging vertical segments can be done similarly.

DEFINITION 12. *An* upward (respectively, downward) hit vertex *h of a horizontal edge e in GG is the farthest of all the obstacle vertices that e can be dragged upward (respectively, downward) to hit without crossing the* interior *of any obstacle. A leftward or rightward hit vertex of a vertical edge in GG is defined symmetrically. The hit vertex of an edge e in GG′ is defined as follows.*

(1) *If e is also in GG then e inherits all its hit vertices in GG.*

(2) *If e is not in GG, then it is an oblique edge. The hit vertices of e are defined to be the hit vertices of the three segments of the* Z-*substitute of e. For each vertical segment g the hit vertex in each direction is the nearest hit vertex of all the edges in GG that compose g.*

Note that a hit vertex of an edge can be undefined if the edge cannot be dragged to hit any vertex or can only be dragged to hit a portion of the obstacle boundary that does not contain any vertex. We attempt to store the Z-substitute for each oblique edge in GG' and compute its hit vertices when we construct GG'. We will embed the computation of the hit vertices of all the edges on GG' in the recursive steps where the edges of GG' are constructed. Following is the algorithm for constructing GG' including the computation of hit vertices. Graph GG and the hit vertices of its edges are generated as intermediate products. The reader is referred to [3], [14] for details of the original algorithm. Here we only put emphasis on how we compute the hit vertices.

Algorithm FindGG′.

1. Sort all vertices in $V' = V \bigcup \{s, d\}$ vertically and horizontally.
2. Find the median vertical cut line L that divides V' in halves.
3. Find the projection points, if visible, from all the vertices to the cut line L. Call those projection points the *potential Steiner points*. Call the edges connecting vertices to their projections the *potential horizontal edges*. An obstacle edge that is cut by L is marked as a potential horizontal edge with a distinction as to whether it borders the obstacle below or above. For two vertices on the same side of L and with the same ordinate, we only generate one potential Steiner point and one potential horizontal edge from the one nearer to L, if it exists. (Note that these potential points and edges are in GG. They are called *potential* since not all of them will be inserted into GG'.)
4. (Find the hit vertices of the potential horizontal edges.)
   Initialize an empty stack for storing those potential horizontal edges that are to the left of L. Scanning these edges from top to bottom, we compute the downward hit vertices. Let the edge we encounter be e and let ve be the vertex that produces e (Fig. 10(a)). Let the edge at the top of the stack be te.
   (a) e is longer than te:
      (a.1) If e borders an obstacle boundary below it, then pop all the edges off from the stack leaving the hit vertices of them undefined, and let ve be the downward hit vertex of e.
      (a.2) Otherwise, push e onto the stack.

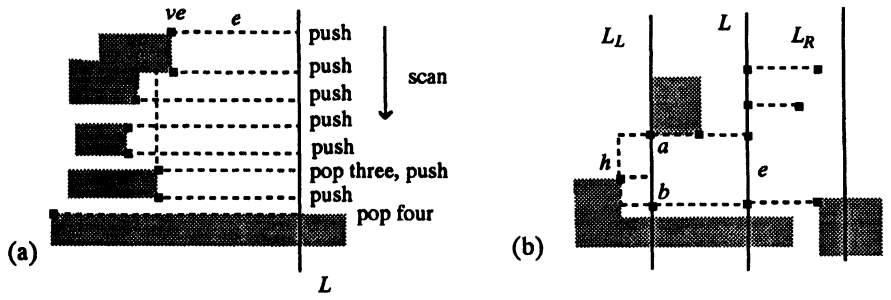


FIG. 10. *Finding the hit vertices for the potential edges.*

(b) e is not longer than te:

(b.1) If e borders an obstacle boundary below it, then keep popping an edge te off from the stack until the stack is empty. Let ve be the downward hit vertex of te if te is not shorter than e and let the hit vertex of te be undefined, otherwise. Let the downward hit vertex of e be ve.

(b.2) Otherwise, keep popping an edge te off from the stack as long as te is longer than e, and let the hit vertex of te be ve. Push e onto the stack.

Find the upward hit vertices in a similar way. Do the same for the potential horizontal edges on the right of L. Call the edges connecting those consecutive potential Steiner points on L the *potential vertical edges*.

5. (Find the hit vertices of the potential vertical edges.)

At each recursive step, we compute the hit vertices of all the potential vertical edges generated on the cut lines. Let L be the cut line under consideration. We look at the two nearest existing cut lines on each side of L, which were generated in previous recursive steps. Let them be L_L and L_R which are on the left side and right side of L, respectively (Fig. 10(b)).

(a) If there is no previous cut line on either side, i.e., the first cut line and those leftmost and rightmost cut lines during the recursive cutting process, then we simply scan over all the points on that side and find the leftward or rightward hit vertices directly. Otherwise, do the following.

(b) The leftward hit vertices of the edges on L are obtained either from vertices between L and L_L, if it exists, or from the leftward hit vertices on the potential vertical edges on L_L, if there is no vertex between L and L_L. The rightward hit vertices of all the potential vertical edges on L can be found in a similar way (from the vertices between L and L_R and those rightward hit vertices on L_R.)

6. (Find the hit vertices of the edges in GG'.)

Cut the plane into horizontal strips of size $O(\sqrt{\log m})$ each and decide which potential Steiner points and which potential edges are added to GG' (see [3], [14]). For those potential edges that are added to GG', let them have the same hit vertices. Some oblique edges which connect pairs of points on different sides of L are generated in each recursive step. We record with each oblique edge, its Z-substitute which consists of three segments: two potential horizontal edges from its end points to L and one vertical segment which is composed of all consecutive potential vertical edges between the potential Steiner points on L of the two end points. We regard the three segments in a Z-substitute as three edges while dragging and thus we need to compute the hit vertices of them respectively. The hit vertices of the two horizontal edges are those computed in step 4. The hit vertex of the vertical segment is obtained by comparing the hit vertex of each individual potential vertical edge obtained in step 5 and selecting the nearest.

7. Do steps 2–6 recursively to the vertex sets on the left side and the right side of L, respectively.

Algorithm FindGG' generates the same graph structure as in [3]. In addition, the hit vertices of all the horizontal and vertical edges on GG' and of the Z-substitutes of all the oblique edges are computed.

LEMMA 4.9. *Algorithm FindGG' runs in $O(m \log^{3/2} m)$ time and space and the hit vertices of all the edges in GG' are correctly computed.*

Proof. The complexity is the same as Algorithm C2 since in each recursive step we compute the hit vertices in time linear in the number of points processed. The only step we need to address is the computation of Z-substitutes and their hit vertices for each pair of vertices inside a strip. Inside each strip, as the accumulated distances are computed [3], [14], [10], we construct a hit vertex table recording the nearest hit vertex in each direction between every pair (not necessarily consecutive) of potential Steiner vertices on L in the strip. This information can later be used for computing the hit vertices of a vertical segment in a Z-substitute. The time complexity therefore is still kept within the same bound. For the correctness of computing the hit vertices, we distinguish two cases as follows:

(1) When a vertex h is the downward hit vertex of a potential horizontal edge e:
If there is any other potential horizontal edge e' between h and e, it must not be shorter than e. If e' is as long as e then it must not border an obstacle below. According to the algorithm, for these cases, e' will be pushed onto a stack. When h is scanned, we pop all edges on top of the stack before we meet an edge shorter than the edge from h to L, which certainly includes e. The case when h is the upward hit vertex of e is proved similarly.

(2) When a vertex h is the leftward hit vertex of a potential vertical edge e:
Let e be on a cut line L. If h falls between L and L_L, then we can certainly find it during the scan. If h falls to the left of L_L, then h must be a hit vertex of some potential vertical edge on L_L. We can assume that h has been recorded as the leftward hit vertex of this potential vertical edge. The reason is that since e can be dragged over L_L to hit h, the two vertices that have projections on the endpoints of e must have projections on L_L as well. These two projections are two potential Steiner points on L_L (denoted as a and b in Fig. 10(b)). Therefore, there must be one or more potential vertical edges on L_L between a and b, and one of them will hit h when dragged leftward. Hence by scanning all the hit vertices on L_L and vertices between L_L and L, we can correctly compute the hit vertices of edges on L. The proof for computing the rightward hit vertices is similar. □

After we have computed all the hit vertices of all the edges of the new guidance graph, the dragging operations can be redefined as follows.

DEFINITION 13. *The dragging operations are defined to be those specified in Definitions* 10 *and* 11 *and the following corresponding operations*:

(1) *The upward hit vertex of w_{new} is the lower one of the upward hit vertices of w and e.*

(2) *The same as in Definition* 10.

(3.1) *The upward hit vertex of w_{new} is the upward hit vertex of e.*

(3.2.1) *If the upward hit vertex of h is higher than q then we drag w to q and let it be w_{new}. The upward hit vertex of w_{new} is that of w.*

(3.2.2) *The same as in Definition* 10.

(4) *and* (5) *are the same as in Definition* 11.

One can see that such dragging operations have the same effects as do the previously defined ones. However, we do not perform segment dragging queries any more. The algorithm is the same except the dragging operations are modified. The space needed is $O(|E_{GG'}|)$ for storing GG'.

THEOREM 4.10. *Problems MBSP, SMBP, and MCP can be solved in* $O(m \log^{3/2} m)$ *time and space.*

5. Conclusion. In this paper, we have presented two results improving on previous ones on finding rectilinear paths among obstacles. We have shown that a *smaller* shortest-path-preserving graph for shortest rectilinear paths among rectilinear obstacles is sufficient and thus obtained a more efficient algorithm for problem *sp*. As a by-product, a faster algorithm for finding the minimum spanning tree of a set of terminals among obstacles is also obtained. Furthermore we have presented a dynamic-searching approach which computes optimal paths dynamically while searching on a guidance graph. Problems MBSP, SMBP, and MCP can be solved efficiently using this approach. It is not clear whether a *better* guidance graph can be obtained to yield a more efficient algorithm for these problems. The problems of how one can improve the time complexities of these algorithms remain open. The results of this paper cannot be extended to non-rectilinear cases in an obvious way.

Acknowledgment. The authors thank the anonymous referee for the comments that greatly helped improve the presentation of the paper.

REFERENCES

[1] A. AHO, J. HOPCROFT, AND J. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
[2] B. CHAZELLE, *An Algorithm for segment dragging and its implementation*, Algorithmica, 3 (1988), pp. 205–221.
[3] K. L. CLARKSON, S. KAPOOR, AND P. M. VAIDYA, *Rectilinear shortest paths through polygonal obstacles in* $O(n \log^{3/2} n)$ *time*, unpublished manuscript, 1989.
[4] P. J. DEREZENDE, D. T. LEE, AND Y. F. WU, *Rectilinear shortest paths with rectangular barriers*, Discrete Comput. Geom., 4 (1989), pp. 41–53.
[5] M. L. FREDMAN AND R. E. TARJAN, *Fibonacci heaps and their uses in improved network optimization algorithms*, J. Assoc. Comput. Mach., 34 (1987), pp. 596–615.
[6] H. IMAI AND T. ASANO, *Dynamic segment intersection search with applications*, Proc. 25th IEEE Symp. on Foundations of Computer Science, Singer Island, Florida, 1984, pp. 393–402.
[7] R. C. LARSON AND V. O. LI, *Finding minimum rectilinear distance paths in the presence of barriers*, Networks, 11 (1981), pp. 285–304.
[8] D. T. LEE, *Proximity and reachability in the plane*, Ph.D. Dissertation, Department of Computer Science, University of Illinois, 1978.
[9] D. T. LEE AND F. P. PREPARATA, *Euclidean shortest paths in the presence of rectilinear barriers*, Networks, 14 (1984), pp. 393–410.
[10] D. T. LEE, C. D. YANG AND T. H. CHEN, *Shortest rectilinear paths among weighted obstacles*, Internat. J. Comput. Geom. Appl., 1 (1991), pp. 109–124.
[11] J. S. B. MITCHELL, *Shortest rectilinear paths among obstacles*, Algorithmica, 8 (1992), pp. 55–88.
[12] ———, *An optimal algorithm for shortest rectilinear path among obstacles*, presented in the First Canadian Conference on Computational Geometry, Montreal, Canada, Aug. 1989.
[13] T. OHTSUKI, *Gridless routers — New wire routing algorithm based on computational geometry*, International Conference on Circuits and Systems, China, 1985.
[14] P. WIDMAYER, *Network design issues in VLSI*, unpublished manuscript, 1989.
[15] Y. F. WU, P. WIDMAYER AND C. K. WONG, *A faster approximation algorithm for the Steiner problem in graphs*, Acta Informatica, 23 (1986), pp. 223–229.
[16] Y. F. WU, P. WIDMAYER, M. D. F. SCHLAG, AND C. K. WONG, *Rectilinear shortest paths and minimum spanning trees in the presence of rectilinear obstacles*, IEEE Trans. Comput., C-36 (1987), pp. 321–331.
[17] C. D. YANG, D. T. LEE AND C. K. WONG, *On bends and lengths of rectilinear paths: A graph-theoretic approach*, Internat. J. Comput. Geom. Appl., 2 (1992), pp. 61–74.
[18] ———, *On bends and distances of paths among obstacles in two-layer interconnection model*, IEEE Trans. Comput., Vol. 43, No. 6, June 1994, pp. 711–724.

SIAM J. COMPUT.
Vol. 24, No. 3, pp. 473–483, June 1995

004

EVALUATION OF POLYNOMIALS USING THE STRUCTURE OF THE COEFFICIENTS*

JÜRG GANZ†

Abstract. A new algorithm to evaluate polynomials that exploits the structure of their coefficients is proposed. This algorithm is an extension of one due to Savage with the advantage that it is not restricted to coefficients from a set whose cardinality is small compared to the degree of the polynomial. The new algorithm is shown to be asymptotically optimum for evaluating arbitrary functions in finite fields and for evaluating polynomials with real coefficients in a binary fixed-point representation. To illustrate that it is nonasymptotically useful as well, the new algorithm is shown to reduce the time for syndrome calculation for binary Bose–Chaudhuri–Hocquenghem (BCH) codes of practical interest.

Key words. polynomial evaluation, straight-line algorithm, finite fields, binary fixed-point representations, decoding BCH codes

AMS subject classifications. 68Q25, 12Y05, 13B25, 94B35

1. Introduction.

This paper is concerned with the evaluation of polynomials

$$p(X) = c_n \cdot X^n + \cdots + c_1 \cdot X + c_0$$

whose coefficients c_i belong to a field $\mathcal{F}$ when the argument X assumes a value x in a field $\mathcal{E}$, which in general is an extension of $\mathcal{F}$. Three different problems can be distinguished, namely,

1. Evaluate $p(X)$ with specified coefficients $c_0, \ldots, c_n$ at an arbitrary argument x.
2. Evaluate $p(X)$ with arbitrary coefficients $c_0, \ldots, c_n$ at a specified argument x.
3. Evaluate $p(X)$ with arbitrary coefficients $c_0, \ldots, c_n$ at an arbitrary argument x.

The "specified" items are those known in advance and can be used to design an appropriate algorithm, whereas the "arbitrary" items are the actual input to the algorithm. The well-known Horner's rule (cf. [3, p. 467])

$$p(X) = (\cdots (c_n \cdot X + c_{n-1}) \cdot X + \cdots + c_1) \cdot X + c_0$$

suggests an algorithm that solves the third problem with $2n$ operations. Of course this algorithm solves the first and second problem as well when the "specified" items are included within the algorithm rather than given as inputs. Horner's rule is easy to implement and needs no storage beyond an accumulator.

In many applications some specified polynomial has to be evaluated many times, which corresponds to the first problem. Here it may be worthwhile to do some precomputation to devise an algorithm suited for that polynomial. The concept of a straight-line algorithm (SLA) has been introduced to study this problem (see also [4] or [3, p. 475] where an SLA is called a polynomial chain). An SLA is defined as follows:

> The input x, which can be any element of $\mathcal{E}$, is assigned as the "result" of step 0 of the algorithm. At each step i, for $i = 1, 2, \ldots, k$, the algorithm performs one operation on two operands. The operation must be either addition or multiplication. One of the two operands at step i must be the result of the operation at step j for some $0 \leq j < i$. The other operand at step i must be either a fixed specified element of $\mathcal{F}$ (and these specified elements will be called the *constants* of the SLA) or the result of the operation at step h for some $0 \leq h \leq j$.

*Received by the editors February 3, 1992; accepted for publication (in revised form) November 11, 1993.

†Signal and Information Processing Laboratory, Swiss Federal Institute of Technology, CH-8092 Zürich, Switzerland.

Note that the choice of operands at each step in an SLA is independent of the *values* of the input and of the *values* computed in previous steps. An SLA *evaluates* a polynomial $p(X)$ if for some i, $0 \leq i \leq k$, and for every input $x \in \mathcal{E}$, the result of step i is $p(x)$. Note that the same SLA can evaluate many different polynomials.

There have been several suggestions for devising SLAs to evaluate polynomials with no restrictions on the coefficients. But the resultant SLAs do not significantly improve on Horner's rule. In fact, it has been shown that there are polynomials such that every SLA that evaluates them needs at least about n additions and $\frac{n}{2}$ multiplications; cf. [3, pp. 475–479] for more details.

In applications where the field $\mathcal{E}$ is a proper extension of $\mathcal{F}$, i.e., $\mathcal{E} \neq \mathcal{F}$, a *nonscalar multiplication*, i.e., a multiplication with both operands in $\mathcal{E}$, is typically more complex than a *scalar multiplication*, i.e., a multiplication with one operand in $\mathcal{F}$ and one operand in $\mathcal{E}$. Thus, SLAs that minimize the number of nonscalar multiplications are of interest. Some SLAs have been proposed that need only about $\sqrt{n}$ nonscalar multiplications, but the total number of operations is still about $2n$ (cf. [6]).

A quite different approach has been taken by Savage [7], who gave an SLA that outperforms previous SLAs when the number $\mathbf{s}$ of different coefficients is much smaller than the degree of the polynomial n. Under this condition, Savage showed that the number of nonscalar multiplications needed is not larger than $2\sqrt{n+1}-2$ and that this can be achieved with a total of only $(n/\log_{\mathbf{s}} n) \cdot (2 + o(1))$ operations, where $o(1)$ denotes a function that is negligible compared to 1 for n large. It is possible to choose the parameters of Savage's algorithm differently so as to minimize the total number of operations. This minimization leads to a version of Savage's algorithm that needs only a total of $(n/\log_{\mathbf{s}} n) \cdot (1 + o(1))$ operations if $\mathbf{s}$ is much smaller than n.

In this paper we extend Savage's result by allowing coefficients c_i from a subset of $\mathcal{F}$, which can be much larger than in the original approach. These coefficients c_i are constrained to be a linear combination of some specified generators $\beta_1, \ldots, \beta_m \in \mathcal{F}$, where all scalars of the linear combination are in a specified set $\mathcal{A} \subseteq \mathcal{F}$ with cardinality $|\mathcal{A}| = a$. Equivalently, the coefficients c_i are constrained to lie in the set

$$\mathcal{S} = \{\beta_1 \cdot a_1 + \cdots + \beta_m \cdot a_m \mid a_j \in \mathcal{A}\} \subseteq \mathcal{F}.$$

Such coefficient sets occur frequently in practical applications. We will also assume with no real loss of generality that $\mathcal{A}$ contains at least the two neutral elements 0 and 1 of the field $\mathcal{F}$. There are at most $s \triangleq a^m \geq |\mathcal{S}|$ different coefficients of the above form. The new SLA given in §2 will evaluate polynomials of degree n with such coefficients using a total of $(n/\log_s n) \cdot (1 + o(1))$ operations under only the restriction that $a = |\mathcal{A}|$ is much smaller than mn. Alternatively, the number of nonscalar multiplications can be upper-bounded by $2\sqrt{n+1}-2$ together with a total of $(n/\log_s n) \cdot (2 + o(1))$ operations under the restriction that a is much smaller than $m^2 n$. These are the same asymptotic forms as for Savage's algorithm except that the upper bound $s = a^m$ on the number of different coefficients has replaced the number $\mathbf{s}$ of different coefficients in Savage's formulas and, more importantly, that the range of validity of the result is now determined by a instead of by $\mathbf{s}$. It follows that the new SLA algorithm outperforms all previous SLAs for all $s \leq n + 1$ and n sufficiently large compared to a. Note that the case $s = n + 1$ is that of polynomials whose coefficients are all different. Further, note that although we demanded that the coefficients must be chosen from a finite set $\mathcal{S}$, we did not restrict the field $\mathcal{E}$ itself to be finite.

In §3, we use counting arguments to show that there are polynomials of degree n with coefficients from any set $\mathcal{S}$ of size $s \leq n+1$ that cannot be evaluated with $(n/\log_s n)\cdot(1-o(1))$ operations by an SLA using at most s constants. This shows the optimality of the new algorithm

in certain circumstances. For evaluating arbitrary functions in a finite field GF(p^m) (with p prime), the new algorithm is shown to be asymptotically optimum as m gets large. Furthermore, the new algorithm is shown to be well suited for evaluating polynomials with real coefficients restricted to a binary fixed-point representation, which is a case of considerable practical interest.

In §4, the new algorithm is applied to the third polynomial-evaluation problem in which both the argument and the coefficients are inputs. It has been shown that Horner's rule is optimum among SLAs with $n+2$ inputs for solving this third problem, cf. [3, p. 479]. Nevertheless, it is shown that the new algorithm can be "extended" by allowing *indirect addressing of operands by inputs* and will then outperform Horner's rule when the coefficients are from the set $\mathcal{S}$ defined above. With a simple adaptation, the new extended algorithm also solves the second polynomial-evaluation problem. This has interesting practical consequences because the computation of a single component of the discrete Fourier transform (DFT) at a fixed frequency x for an arbitrary time sequence of the form $c_0, \ldots, c_n$ corresponds exactly to this second problem. Finally, an application in decoding binary Bose–Chaudhuri–Hocquenghem (BCH) codes is given that shows that the new extended algorithm improves upon known methods of syndrome calculation for parameters far away from asymptotics (e.g., for the 7-error-correcting binary BCH code with block length 127).

In the last section, the similarities and differences between the new algorithm and the one due to Savage are discussed. Some possible generalizations of the new algorithm are also given.

2. The new algorithm. In this section, we propose a new SLA to solve the first polynomial-evaluation problem. Assume with no real loss of generality that $n = q \cdot p \cdot l - 1$, so that the polynomial $p(X)$ of degree n with coefficients in $\mathcal{S}$ can be written as

$$p(X) = g_0(X) + g_1(X) \cdot X^{pl} + \cdots + g_{q-1}(X) \cdot X^{(q-1)pl}, \tag{1}$$

where $g_k(X)$ is a polynomial of degree less than pl with coefficients in $\mathcal{S}$. Let the equation

$$g_k(X) = \beta_1 \cdot h_{k,1}(X) + \cdots + \beta_m \cdot h_{k,m}(X) \tag{2}$$

define the polynomials $h_{k,1}(X), \ldots, h_{k,m}(X)$, which are polynomials of degree less than pl with coefficients in $\mathcal{A}$, and similarly let

$$h_{k,j}(X) = t_{k,j,0}(X) + t_{k,j,1}(X) \cdot X^l + \cdots + t_{k,j,p-1}(X) \cdot X^{(p-1)l} \tag{3}$$

define the polynomials $t_{k,j,0}(X), \ldots, t_{k,j,p-1}(X)$ with coefficients in $\mathcal{A}$ and degree less than l.

Given the input $x \in \mathcal{E}$, the algorithm comprises the following four steps:

Step 1. Form p lists containing all possible $t_{k,j,i}(x) \cdot x^{il}$ and compute x^{pl}:

a) Form list 0 of the values of all a^l polynomials of degree less than l with coefficients in $\mathcal{A}$ evaluated at x. (This requires $l-2$ nonscalar multiplications to form $x^2, x^3, \ldots, x^{l-1}$, together with $(l-1)(a-2)$ scalar multiplications to evaluate the monomials with coefficients not 1, together with $a^l - (a-1)l - 1$ additions to evaluate the binomials, trinomials, etc., by summing values already computed, since each addition can be designed to evaluate a new polynomial in the manner that the value of a trinomial is the sum of the value of a monomial and a binomial, etc.)

b) For $i = 1, 2, \ldots, p-1$, form list i consisting of the values in list 0 each multiplied with x^{il}. (This requires, for each value of i, l nonscalar multiplications starting from $x^{il-1} = x^{(i-1)l+l-1}$ to form $x^{il}, x^{il+1}, \ldots, x^{il+l-1}$,

together with $l(a-2)$ scalar multiplications to evaluate the monomials, together with $a^l-(a-1)l-1$ additions to evaluate the binomials, trinomials, etc., by summing values already computed in a way similar to that in Step 1a.)

c) Compute the value of x^{pl}. (This requires only one nonscalar multiplication because $x^{pl-1}=x^{(p-1)l+l-1}$ is already available in list $p-1$.)

(The whole of Step 1 requires a total of less then $p\cdot a^l$ operations, of which $pl-1$ are nonscalar multiplications.)

Step 2. Compute $h_{k,j}(x)$ for $0\le k<q$ and $1\le j\le m$ by adding according to (3) the values $t_{k,j,i}(x)\cdot x^{il}$ from list i for $i=0,1,\dots,p-1$. (This takes at most $p-1$ additions for each of the qm values of k and j.)

Step 3. Compute $g_k(x)$ for $0\le k<q$ by first multiplying $h_{k,j}(x)$ by the scalar β_j and then summing for $j=1,2,\dots,m$ according to (2). (This takes m scalar multiplications and $m-1$ additions for each of the q values of k.)

Step 4. Compute $p(x)$ according to (1) in the manner (reminiscent of Horner's rule)

$$p(x)=(\cdots(g_{q-1}(x)\cdot x^{pl}+g_{q-2}(x))\cdot x^{pl}+\cdots+g_0(x).$$

(This takes $q-1$ nonscalar multiplications and $q-1$ additions.)

The entire algorithm requires

$$\begin{aligned}C_{tot}&<p\cdot a^l+q\cdot m\cdot(p-1)+q\cdot(2m-1)+2\cdot(q-1)\\&=p\cdot a^l+q\cdot m\cdot p+q\cdot m+q-2\end{aligned}$$

operations, of which

$$C_{ns}=p\cdot l+q-2$$

are nonscalar multiplications. We optimize these numbers by the following choices of the parameters:

1. The bound on the total number of operations C_{tot} is approximately minimized by choosing

$$q=\frac{n+1}{pl},\quad l=\lceil\log_a(mn)-3\cdot\log_a\log_a(mn)\rceil,\quad p=\lceil\log_a(mn)\rceil.$$

When a is much smaller than mn, this gives

$$C_{tot}\le\frac{mn}{\log_a(mn)}\cdot(1+o(1))\le\frac{n}{\log_s n}\cdot(1+o(1)),$$

where $o(1)$ is negligible compared to 1 for n large. This total computation is less than that of Horner's algorithm if the set $\mathcal{S}$ of allowed coefficients satisfies

$$|\mathcal{S}|\le s=a^m\le n+1.$$

Note that this restriction still permits all $n+1$ coefficients of the polynomial to be different. This removes the severe restriction that the number of different coefficients must be much smaller than the degree n of the polynomial, which is needed to make Savage's algorithm [7] superior to Horner's.

2. The number of nonscalar multiplications C_{ns} is minimized by choosing

$$q=pl=\sqrt{n+1}.$$

Independently of a and m, this gives

$$C_{ns} = 2\sqrt{n+1} - 2.$$

Fixing this choice of q, the bound on the total number of operations is then approximately minimized by the choice

$$l = \lceil \tfrac{1}{2} \log_a(m^2 n) - \log_a \log_a(m^2 n) \rceil.$$

For a small compared to $m^2 n$, this yields

$$C_{tot} \leq \frac{mn}{\log_a(m^2 n)} \cdot (2 + o(1)) \leq \frac{n}{\log_s n} \cdot (2 + o(1)),$$

which again is superior to other SLAs when $|\mathcal{S}| \leq s \leq n + 1$.

For software implementations of the algorithm, we consider briefly the amount of storage needed when the total number of operations is optimized. The necessary storage is determined by Step 1 and, for the optimizing choice of parameters, equals storage for

$$p \cdot a^l \approx \frac{mn}{(\log_a(mn))^2} \leq \frac{n}{m \cdot (\log_s n)^2}$$

elements of $\mathcal{E}$. On the average, each of these elements is used

$$\frac{qm}{a^l} \approx \log_a(mn) \approx m \cdot \log_s n$$

times in Step 2.

Note that Steps 3 and 4 can essentially be executed in reverse order as follows:

Step 3′. Compute $g'_j(x)$ for $1 \leq j \leq m$ in the manner

$$g'_j(x) = (\cdots (h_{q-1,j}(x) \cdot x^{pl} + h_{q-2,j}(x)) \cdot x^{pl} + \cdots + h_{0,j}(x).$$

(These are polynomials with coefficients in $\mathcal{A}$ and degree at most n. The computation takes $q - 1$ nonscalar multiplications and $q - 1$ additions for each of the m values of j.)

Step 4′. Compute $p(x)$ by first multiplying $g'_j(x)$ by the scalar β_j and then summing for $j = 1, 2, \ldots, m$. (This takes m scalar multiplications and $m - 1$ additions.)

This exchange reduces the total number of operations by q, but has no effect on the asymptotics and significantly increases the number of nonscalar multiplications.

Finally, we want to stress that we designed an SLA that uses as constants the elements $\beta_1, \ldots, \beta_m$ and the elements of $\mathcal{A}$. The specific coefficients of the polynomial determine the exact structure of the algorithm.

3. Optimality of the new algorithm. In this section, it is shown for $|\mathcal{S}| = s$ that the new algorithm is asymptotically optimum among all SLAs that use constants from a set of size $s = |\mathcal{S}|$ (which need not be the set $\mathcal{S}$ itself). First, we count the number of different polynomials that can be evaluated by SLAs of length k that use constants from a set of size s. For this purpose, consider a gate network having as input signals x and the constants $\gamma_1, \ldots, \gamma_s$ and containing k gates, each of which performs the operation of addition or multiplication, cf. Fig. 1.

We now specify that the first input to each gate is either the input signal x or the output of another gate and that the second input to each gate is either an input signal (i.e., x or a constant γ_i) or the output of another gate. For each gate there are 2 choices for its type, k

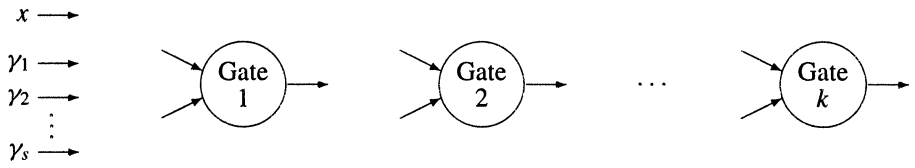


FIG. 1. *The components of the gate networks used to bound the length of SLAs.*

choices for its first input, and $k+s$ choices for its second input. This gives exactly $(2\cdot k\cdot(k+s))^k$ gate networks as defined here.

We will call a gate network irredundant if it does not contain two gates of the same type with identical inputs. We will say that two gate networks are essentially the same if they differ only by a permutation of the gate labels. There are exactly $k!$ gate networks that are essentially the same as a given irredundant gate network. Thus the number of essentially different irredundant gate networks is upper-bounded by the total number of gate networks divided by $k!$, i.e., there are at most $(2\cdot k\cdot(k+s))^k/k!$ essentially different irredundant gate networks. A gate network performs a well-defined computation only if its connections are loop free, but we do not need to exploit this fact.

Every SLA corresponds to a unique gate network in the manner that gate i implements step i of the SLA. We will call an SLA irredundant if its corresponding gate network is irredundant, i.e., if it contains no step that merely repeats the computation of a previous step. Similarly, we will call two SLAs essentially different if their corresponding gate networks are essentially different. It follows that there are at most $(2\cdot k\cdot(k+s))^k/k!$ essentially different irredundant SLAs of length k. (It is important to note that the $k!$ irredundant gate networks that are essentially the same as the gate network corresponding to some irredundant SLA do not all correspond to SLAs because some permutations of the gate labels may require that a gate with label i operates on the output of a gate with label j where $j > i$, so that the corresponding "algorithm" would not be an SLA. Thus, in general there are fewer than $k!$ irredundant SLAs of length k that are essentially the same.)

Every SLA of length k evaluates at most k different polynomials, one at each step. Moreover, every polynomial that can be evaluated by an SLA of length k can trivially also be evaluated by an irredundant SLA of length k. Because SLAs that are essentially the same evaluate the same set of polynomials, it now follows that the number P_k of polynomials that can be evaluated by SLAs of length k satisfies

$$\begin{aligned} P_k &\le k\cdot(2\cdot k\cdot(k+s))^k/k! \\ &\le (k+s)^k\cdot(2e)^k\cdot\sqrt{k}, \end{aligned} \tag{4}$$

where we have used Stirling's bound on the factorial [2, pp. 52–54].

For all of the s^{n+1} polynomials of degree $n \ge s-1$ with coefficients in a set of size s to be evaluated by SLAs of length k would require that

$$P_k \ge s^{n+1}$$

or, equivalently, that

$$\log_s P_k \ge n+1. \tag{5}$$

Suppose that k satisfies

$$k \le \frac{n}{\log_s n}\cdot\left(1-\frac{\log_s 4e}{\log_s n}\right) < n, \tag{6}$$

where we note that there exists an n_0 such that the middle member in these inequalities is greater than 1 for all $n \geq n_0$ and any s satisfying $2 \leq s \leq n+1$. Taking the logarithm of both sides in (4), upperbounding $k+s$ by $2n$, and then replacing k by its tighter upperbound in (6) gives

$$\begin{aligned} \log_s P_k &\leq k(\log_s n + \log_s 4e) + \tfrac{1}{2}\log_s k \\ &\leq n\left(1 - \left(\frac{\log_s 4e}{\log_s n}\right)^2\right) + \tfrac{1}{2}\log_s n \\ &< n \end{aligned}$$

for $n \geq n_0$, which contradicts (5). We conclude that, for large enough n and any set of size s with $2 \leq s \leq n+1$, there are polynomials of degree n and coefficients in this set that cannot be evaluated by an SLA with constants from a set of size s that uses

$$k = \frac{n}{\log_s n} \cdot (1 - o(1))$$

or fewer operations (where o(1) is negligible compared to 1 for n large). It follows that the new algorithm—designed for certain sets of coefficients of size s—is asymptotically optimum.

Counting arguments similar to those above were used for similar applications by Savage [8, pp. 114–115] and Wegener [10, pp. 87–90]. These authors did not introduce the concept of irredundant gate networks but simply claimed that the number of essentially different gate networks is upper-bounded by the total number of gate networks divided by $k!$, which is in fact not true. The counting argument given here shows, however, that the bounds in [8, pp. 114–115] and [10, pp. 87–90] on the required number of gates are nonetheless correct.

Application 1: Finite fields. Consider the finite field GF(p^m) with p prime. It is well known that every function f : GF(p^m) $\longrightarrow$ GF(p^m) may be represented by a polynomial of degree n less than p^m with coefficients in GF(p^m). Therefore, the number of operations needed to evaluate any function can be upper-bounded by that needed for the evaluation of polynomials with degree $n = p^m - 1$. The new algorithm can be applied with $\mathcal{A} = \text{GF}(p)$ and $\beta_1, \ldots, \beta_m$ forming a basis for GF(p^m) over GF(p); thus $\mathcal{S} = \mathcal{F} = \mathcal{E} = \text{GF}(p^m)$. The condition that $|\mathcal{A}| = p$ is much smaller than $mn = m(p^m - 1)$ is certainly fulfilled for large enough m. It follows that, for large m, every such function f may be evaluated with at most

$$C_{tot} \leq p^m \cdot (1 + o(1))$$

operations in GF(p^m), where $o(1)$ is negligible compared to 1.

On the other hand, because $s = |\mathcal{S}| = p^m$, the previous arguments showed that there are functions (i.e., polynomials of degree less than p^m) that require more than

$$p^m \cdot (1 - o(1))$$

operations to be evaluated in GF(p^m) (see also [4] where this same bound is shown to apply when subtraction and division are also allowed as operations). We conclude that the new algorithm is asymptotically optimum for evaluating arbitrary functions in GF(p^m) when m is large.

Application 2: Binary fixed-point representation. Many practical computations use a binary fixed-point representation of real numbers. This representation can be described in our notation by

$$\mathcal{A} = \{0, 1\} \text{ and } \beta_i = 2^{i+offset}$$

for $i = 1, \ldots, m$, where m gives the precision and *offset* determines the binary point of the representation. This defines the set of coefficients that are allowed. The operations may be realized with greater precision, or even in floating point, in order to achieve the desired precision of the result. Considering the results of §2, we see that the new algorithm may be preferable for polynomials of degree 2^m with coefficients in this representation, and it certainly outperforms other known SLAs for large degrees. In fact, for the total number of operations, the new algorithm is asymptotically optimum among all SLAs using constants in this representation.

4. Adaptation to variable coefficients. The new SLA of §2 evaluates a polynomial $p(X)$ with the argument x as input. We now adapt this algorithm to the case where the values of the coefficients $c_0, \ldots, c_n$ are additional inputs. The new algorithm can be adapted to solve this third polynomial-evaluation problem when the following conditions are fulfilled:

1. The coefficients $c_0, \ldots, c_n$ lie in a set $\mathcal{S}$ for which $\beta_1, \ldots, \beta_m$ and the elements of $\mathcal{A}$ are known in advance. (This allows one to perform the calculations of Steps 1 and 3 in exactly the same way as before.)
2. Given a coefficient $c_i = \beta_1 \cdot a_{i,1} + \cdots + \beta_m \cdot a_{i,m}$, it must be easy to compute its components $a_{i,1}, \ldots, a_{i,m}$ (or, as would generally be the case in practice, these components are the actual inputs to the algorithm).
3. The components $a_{i,j}$ of the coefficients c_i, which are the inputs to the algorithm, are allowed to determine the operands in Step 2 of the previous algorithm. (This means that the new algorithm is not in fact an SLA, but is rather the result of extending the notion of an SLA to allow *indirect addressing of operands by inputs*.)

The third condition prevents a hard-wired implementation of the new extended algorithm. Nevertheless, this extension is well suited for a software implementation of the algorithm because indirect addressing is easy to realize. In fact, most software implementations of the SLA of §2, in which the components $a_{i,j}$ are constants, would probably also use indirect addressing for convenience. Therefore it makes no great difference in a software implementation whether the components $a_{i,j}$ are constants or inputs. If the implementation is on a general-purpose single-processor computer, then the number of field operations needed essentially determines the performance of the algorithm. In this case, all the counting and optimizing of the number of operations for the SLA of §2 are still valid for the extension. This also includes the computation of the amount of storage needed, which was determined at the end of §2. It follows that the extended new algorithm can outperform Horner's rule even for the third polynomial-evaluation problem.

There are also applications where the argument x is fixed and only the coefficients c_i are inputs. To adapt our algorithm to this case, which is the second polynomial-evaluation problem, all the above conditions still must be observed, but there is now the possibility to precompute all the lists of Step 1 because both x and the elements of $\mathcal{A}$ are known in advance. Adapting the new algorithm, we find that the performance strongly depends on the storage D available for the lists of Step 1. With the same parameters as in §2, we find that

$$D = p \cdot a^l$$

storage locations are needed together with

$$C_{tot} = q(pm + m + 1) - 2$$

operations to perform Steps 2, 3, and 4. Again assuming without real loss of generality that $n = l \cdot p \cdot q - 1$, we see that the number of operations can be well approximated even for

moderate n as

$$C_{tot} \approx \frac{m\,n}{l} \cdot \left(1 + \frac{1}{p}\right).$$

Choosing now $p \approx \log_a D$, the storage requirement is equivalent to $l \approx \log_a D - \log_a \log_a D$ and the number of operations becomes

$$C_{tot} \approx \frac{m\,n}{\log_a D} = \frac{n}{\log_s D}$$

for D large compared to a. This shows that the adaptation of the new algorithm to the second polynomial-evaluation problem yields an algorithm that outperforms Horner's rule if storage on the order of the number of possibly different coefficients s is available. Note that at the same time the number of operations needed for the precomputation (which is less than D) and the number of nonscalar multiplications needed in Step 4 ($C_{ns} = q - 1$) are quite small. But the minimizing of C_{ns} is preferably done with a simple algorithm that first precomputes the powers of the argument x. As an example, if $D = n$ storage locations are available, then all needed powers of x can be precomputed and, with n scalar multiplications and n additions, every polynomial can be computed.

Application 3: Decoding binary BCH codes. Binary BCH codes are widely used for error correction or detection; a detailed treatment can be found in [1, Chap. 7]. For a primitive binary BCH code of block length $N = 2^M - 1$ and design distance $d = 2t+1$, the received block $[c_{N-1}, \cdots, c_1, c_0]$ can be identified with the polynomial $p(X) = c_{N-1}X^{N-1} + \cdots + c_1 X + c_0$ with binary coefficients c_i. There exists a decoding algorithm that computes which coefficients are incorrect, provided that at most t errors occurred in transmission. The most commonly used algorithm consists of computing syndromes, finding the error locator polynomial, and calculating its zeros (see [1, pp. 183–193] for all details). In this application we focus on the first part of this decoding algorithm, namely, computing the syndromes $S_j = p(\alpha^j)$ for $j = 1, 3, \ldots, 2t - 1$, where α is a primitive element in GF(2^M).

Consider now an implementation of such a decoding algorithm on a general-purpose single-processor computer. Assume that procedures for executing field operations in GF(2^M) are available. Obviously, the number of field operations required determines the running time of the decoding algorithm. The syndrome calculation consists of evaluating the polynomial $p(X)$ for t values of its argument. This can be done by Horner's rule with $2t(N-1)$ operations. The additions in Horner's rule call for adding a coefficient that is either 0 or 1, and hence these additions can be skipped or easily implemented. In general, however, the $C_s \triangleq t(N-1)$ multiplications in Horner's rule must be performed. Indeed, this is a substantial part of the whole decoding computation.

We propose an algorithm to reduce the number of field operations needed to calculate the syndromes. The syndrome calculation requires the evaluation of a polynomial $p(X)$ of degree $N - 1$ with binary coefficients c_i for t values x of X. This corresponds exactly to the second polynomial-evaluation problem. Because the coefficients are binary, we have $m = 1$ in the definition of the set $\mathcal{S}$ of coefficients. Thus, Step 3 of the algorithm in §2 can be omitted. Therefore, it suffices to perform the following calculations for $j = 1, 3, \ldots, 2t - 1$:

Step 1. Precompute the lists $T_0, T_1, \ldots, T_{p-1}$ containing the values

$$T_i[\underline{a}] = \left(a_{l-1} \cdot (\alpha^j)^{l-1} + \cdots + a_1 \cdot \alpha^j + a_0\right) \cdot (\alpha^j)^{il}$$

for all $\underline{a} = [a_{l-1}, \ldots, a_0] \in \{0, 1\}^l$. Additionally compute $\alpha^{jpl} = \alpha^j \cdot T_{p-1}[[1, 0, \ldots, 0]] = \alpha^j \cdot (\alpha^j)^{(p-1)l+l-1}$. (This can be done in the manner of Step 1 in §2 with less than $p \cdot 2^l$ operations, of which $pl - 1$ are multiplications. The storage needed is equal to the total number of operations.)

Step 2. For the binary inputs $c_0, \ldots, c_{N-1}$, compute

$$h_k(\alpha^j) = T_{p-1}[\underline{c}_{k,p-1}] + \cdots + T_0[\underline{c}_{k,0}],$$

where $\underline{c}_{k,i} = [c_{kpl+il+l-1}, \ldots, c_{kpl+il+1}, c_{kpl+il}]$ for $k = 0, \ldots, q-1$. (This can be done with $q(p-1)$ additions.)

Step 3. Finally, compute

$$p(\alpha^j) = (\cdots (h_{q-1}(\alpha^j) \cdot \alpha^{jpl} + h_{q-2}(\alpha^j)) \cdot \alpha^{jpl} + \cdots + h_0(\alpha^j)$$

by Horner's rule. (This requires $2(q-1)$ operations (half of which are multiplications) where $q = \lceil N/pl \rceil$.)

There is no additional storage needed for the computation in Steps 2 and 3 if the h_k are computed as they are used in Step 3. Table 1 lists the operations required for calculating syndromes for the binary BCH code ($N = 127$, $t = 7$) by this new method. For the chosen values, which are of practical interest, using Horner's rule to compute the syndromes requires $C_s = t(N-1) = 882$ multiplications and 882 trivial additions; the new method requires only 98 multiplications and 308 (nontrivial) additions. The improvement is especially great for the usual implementation of field operations in which multiplication is much more time-consuming than addition. This shows that the use of a small amount of storage can significantly simplify syndrome calculation for practical applications.

The optimum choice of the parameters is highly dependent on the actual implementation of the field operations and on the trade-off between storage and time that is suited to the application. The new algorithm gives an efficient way to use a certain amount of storage to minimize the number of field operations required to calculate the syndromes. If only a very small amount of storage is available, then there is the possibility of doing the precomputation with every computation. For this case, it is sufficient to store the lists $T_0, \ldots, T_{p-1}$ for one j, which amounts to $p2^l$ storage locations. The number of field operations is then the sum of those in the precomputation and the computation, which is still small. For the above parameters, 24 storage locations together with 546 field operations (154 multiplications) suffice.

5. Generalizations and remarks. The new polynomial-evaluation algorithm of §2 uses some ideas from an algorithm of Lupanov [5] (see also [10, pp. 91–92]) for evaluating Boolean functions in many variables. Strassen [9] extended Lupanov's ideas to an algorithm evaluating a multivariable polynomial $p(X_1, \ldots, X_r) \in \mathcal{F}[X_1, \ldots X_r]$, having degree less than n in each variable, at a value $[x_1, \ldots, x_r] \in \mathcal{E}^r$. Strassen showed that if $|\mathcal{F}|$ is much smaller than n^r, then at most $(n^r / \log_{|\mathcal{F}|}(n^r)) \cdot (1 + o(1))$ operations in $\mathcal{E}$ are needed. For $r = 1$, this result is very similar to Savage's [7]; both results hold if the number of different coefficients is much smaller than the degree of the polynomial. But Strassen's result is stated with only a small finite field as the coefficient set, whereas Savage's result holds for any small set. The crucial new idea exploited in this paper is to extend the set of allowed coefficients to the set of restricted linear combinations of specified generators, the restriction being that the scalars

TABLE 1

Computation count for calculating syndromes for the binary BCH code ($N = 127$, $t = 7$) by the new method. (The precomputation requires $t(pl-1) = 56$ multiplications and $t(p(2^l - l - 1)) = 84$ additions.)

Parameters			Precomputation and storage	Computation		
				#Additions	#Mult.	#Operations
p	l	$q = \lceil \frac{N}{pl} \rceil$	$tp2^l$	$t(qp-1)$	$t(q-1)$	$t(qp+q-2)$
3	3	15	168	308	98	406

come from a small set. This makes it possible to improve on Horner's algorithm even in cases where all the coefficients of the polynomial are distinct. This extension can be applied to polynomials in many variables as well.

The specific similarities and differences between the new algorithm and the one due to Savage [7] are the following. Steps 1 and 2 correspond to Savage's "computation of linear forms" and are only slightly refined compared to the steps used in the polynomial-evaluation application of his algorithm. Step 3 exploits our assumed structure of the coefficients, while Step 4 is exactly the same as Savage's final step. For the case where $m = 1$ (i.e., $\mathcal{A} = \mathcal{S}$), Step 3 is not required and the new algorithm virtually reduces to Savage's because our refinements in Step 1 and 2 are asymptotically negligible.

In all of the above, we have never actually used the fact that $\mathcal{F}$ or $\mathcal{E}$ are fields so that division by nonzero elements is well defined. In fact, we have not even used the commutativity of multiplication. Therefore our proposed algorithm works even if $\mathcal{F}$ and $\mathcal{E}$ are distinct noncommutative rings related by a ring-homomorphism $\phi : \mathcal{F} \longrightarrow \mathcal{E}$ with $\phi(1) = 1$, where the 1's are the neutral elements of the multiplication operations in the respective rings. As an example $\mathcal{F}$ could be the ring of 2×2 real matrices, $\mathcal{E}$ the ring of 4×4 real matrices, and ϕ the ring-homomorphism

$$\phi(M) = \begin{bmatrix} M & 0 \\ 0 & M \end{bmatrix},$$

where $M \in \mathcal{F}$ and 0 is the neutral element for addition in $\mathcal{F}$.

Finally, our new algorithm can be easily generalized to handle polynomials with coefficients from a set

$$\mathcal{S} = \{\beta_1 \cdot a_{i,1} + \cdots + \beta_m \cdot a_{i,m} \mid a_{i,j} \in \mathcal{A}_j\},$$

where the finite sets $\mathcal{A}_1, \ldots, \mathcal{A}_m$ are all different. It is possible to show the same asymptotics as before if the cardinalities of the sets are bounded by $|\mathcal{A}_j| \leq a$. We do not carry out this generalization in detail because we do not see a practical application for it at the moment.

Acknowledgments. The author is grateful to J. L. Massey, who significantly improved the presentation of this paper, and to A. Hiltgen for many constructive discussions.

REFERENCES

[1] R. E. BLAHUT, *Theory and Practice of Error Control Codes*, Addison-Wesley, Reading, MA, 1983.
[2] W. FELLER, *An Introduction to Probability Theory and its Applications*, Vol. 1, 3rd ed., Wiley, New York, 1968.
[3] D. E. KNUTH, *The Art of Computer Programming: Volume 2 – Seminumerical Algorithms*, 2nd ed., Addison-Wesley, Reading, MA, 1981.
[4] A. LEMPEL, G. SEROUSSI, AND J. ZIV, *On the power of straight-line computations in finite fields*, IEEE Trans. Inform. Theory, IT-28 (1982), pp. 875–880.
[5] O. B. LUPANOV, *A method of circuit synthesis*, Izv. Vyssh. Uchebn. Zaved. Radiofiz., 1 (1958), pp. 120–140.
[6] M. S. PATERSON AND L. J. STOCKMEYER, *On the number of nonscalar multiplications necessary to evaluate polynomials*, SIAM J. Comput., 2 (1973), pp. 60–66.
[7] J. E. SAVAGE, *An algorithm for the computation of linear forms*, SIAM J. Comput., 3 (1974), pp. 150–158.
[8] ———, *The Complexity of Computing*, Wiley, New York, 1976. (Reprint with corrections by the Robert E. Krieger Pub. Co., Malabar, FL, 1987).
[9] V. STRASSEN, *Berechnungen in partiellen Algebren endlichen Typs*, Computing, 11 (1973), pp. 181–196.
[10] I. WEGENER, *The Complexity of Boolean Functions*, Wiley, New York (Teubner, Stuttgart), 1987.

SIAM J. COMPUT.
Vol. 24, No. 3, pp. 484–493, June 1995

005

WHEN IS THE ASSIGNMENT BOUND TIGHT FOR THE ASYMMETRIC TRAVELING-SALESMAN PROBLEM?*

ALAN FRIEZE†, RICHARD M. KARP‡, AND BRUCE REED§

Abstract. We consider the probabilistic relationship between the value of a random asymmetric traveling salesman problem $ATSP(M)$ and the value of its assignment relaxation $AP(M)$. We assume here that the costs are given by an $n \times n$ matrix M whose entries are independently and identically distributed. We focus on the relationship between $Pr(ATSP(M) = AP(M))$ and the probability p_n that any particular entry is zero. If $np_n \to \infty$ with n then we prove that $ATSP(M) = AP(M)$ with probability 1-o(1). This is shown to be best possible in the sense that if $np(n) \to c$, $c > 0$ and constant, then $Pr(ATSP(M) = AP(M)) < 1 - \phi(c)$ for some positive function ϕ. Finally, if $np_n \to 0$ then $Pr(ATSP(M) = AP(M)) \to 0$.

Key words. traveling salesman, probabilistic analysis

AMS subject classifications. 05C80, 90C27

1. Introduction.

The *assignment problem* (AP) is the problem of finding a minimum-weight perfect matching in an edge-weighted bipartite graph. An instance of the AP can be specified by an $n \times n$ matrix $M = (m_{ij})$; here m_{ij} represents the weight of the edge between x_i and y_j, where $X = \{x_1, x_2, \ldots, x_n\}$ is the set of "left vertices" in the bipartite graph and $Y = \{y_1, y_2, \ldots, y_n\}$ is the set of "right vertices." The AP can be stated in terms of the matrix M as follows: find a permutation σ of $\{1, 2, \ldots, n\}$ that minimizes $\sum_{i=1}^{n} m_{i,\sigma(i)}$. Let $AP(M)$ be the optimal value of the instance of the AP specified by M.

The *asymmetric traveling-salesman problem* (ATSP) is the problem of finding a Hamiltonian circuit of minimum weight in an edge-weighted directed graph. An instance of the ATSP can be specified by an $n \times n$ matrix $M = (m_{ij})$ in which m_{ij} denotes the weight of edge $< i, j >$. The ATSP can be stated in terms of the matrix M as follows: find a cyclic permutation π of $\{1, 2, \ldots, n\}$ that minimizes $\sum_{i=1}^{n} m_{i,\pi(i)}$; here a cyclic permutation is one whose cycle structure consists of a single cycle. Let $ATSP(M)$ be the optimal value of the instance of the ATSP specified by M.

It is evident from the parallelism between the above two definitions that $AP(M) \leq ATSP(M)$. The ATSP is NP hard, whereas the AP is solvable in time $O(n^3)$. Several authors (for a recent survey see [BaTo]) have investigated whether the AP can be used effectively in a branch-and-bound method to solve the ATSP.

The most striking evidence of the power of this approach is given by the recent work of Miller and Pekny [MiPe]. Among many other computational results, they obtained optimal solutions to random instances with up to 500,000 cities, in which the m_{ij} were drawn independently from the integers in the range $[0, n]$. Miller and Pekny noticed that $AP(M)$ was often equal to $ATSP(M)$, and they exploited this observation by developing a special method to search for a cyclic permutation among the optimal solutions to the AP.

Motivated by the computational experience of Miller and Pekny, we have investigated the following question: when the m_{ij} are drawn independently from a common distribution (over, say, the nonnegative reals), what is the probability that $AP(M) = ATSP(M)$? The answer depends on the probability that an entry is zero. We show that, if the expected number of zeros in a row of M tends to infinity as $n \to \infty$ then the probability that $AP(M) =$

*Received by the editors August 5, 1992; accepted for publication (in revised form) November 16, 1993.

†Department of Mathematics, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213. The research of this author was supported by National Science Foundation grant CCR-9024935.

‡University of California, Berkeley and International Computer Science Institute, Berkeley, California 94720. The research of this author was supported by National Science Foundation grant CCR-9017380.

§Department of Mathematics, Carnegie-Mellon University, Pittsburgh, PA 15213.

$ATSP(M)$ tends to 1, and we give an $O(n^3)$-time algorithm for finding an optimal solution to the ATSP with high probability. Conversely, if the underlying distribution is uniform over the range of integers $[0..\lfloor c_n n \rfloor]$, where c_n tends to infinity with n, then the probability that $AP(M) = ATSP(M)$ tends to 0. Finally, we show that if the underlying distribution is uniform over the range of integers $[0..\lfloor cn \rfloor]$ where c is a positive constant, then the probability that $AP(M) = ATSP(M)$ does not tend to 1. We conjecture that for distributions of this type, the probability that $AP(M) = ATSP(M)$ tends to some positive constant less than 1 which depends on c.

The results of this paper are closely related to some earlier results of Karp [K], Karp and Steele [KS], and Dyer and Frieze [DF]. Here the $m_{i,j}$ are drawn independently from the uniform distribution over [0,1]. Karp showed that $ATSP(M)/AP(M) = 1 - o(1)$ (whp) (we use the notation (whp) as shorthand for "with probability tending to 1 as n tends to infinity"). Later, Karp and Steele and then Dyer and Frieze strengthened this result in several ways. For example, the latter paper shows that the error term is $o((\log n)^4/n)$.

2. The theorems.

THEOREM 2.1. *Let $\{X_n\}$ be a sequence of random variables over the nonnegative reals. Let $p_n = Pr[X_n = 0]$ and let $w(n) = np_n$. Let $M = M(n)$ be an $n \times n$ matrix whose entries are drawn independently from the same distribution as X_n. If $w(n) \to \infty$ as $n \to \infty$ then $AP(M) = ATSP(M)$ (whp).*

(Examination of the proof of Theorem 2.1 reveals that the distribution of nonzeros can be more complicated than actually stated. Indeed one can allow the costs to be generated as follows: start with an *arbitrary* real nonnegative $n \times n$ matrix M. Randomly permute its rows and columns. Then for each $i, j \in [n]$ replace $M_{i,j}$ with zero, with probability p_n. There is also the proviso that the probability of two identical columns should tend to zero with n.)

Frieze [Fr] has shown that, if $w(n) = \ln n + t(n)$, where $t(n)$ tends to infinity, then $ATSP(M) = 0$ (whp), and so $AP(M) = ATSP(M)$ (whp). Thus, we restrict attention to the case where $w(n) = O(\ln n)$. The case where X_n has the uniform distribution over the range of integers $[0..N(n)]$ is particularly relevant to the Miller–Pekny computations. In this case, Theorem 2.1 tells us that $AP(M) = ATSP(M)$ (whp) provided that $N(n) = o(n)$.

THEOREM 2.2. *Let $M = M(n)$ be an $n \times n$ matrix whose entries are drawn independently from the uniform distribution over $\{0, 1, \ldots, \lfloor cn \rfloor\}$, where c is a positive constant. Then the probabilty that $AP(M) \neq ATSP(M)$ does not tend to zero as n tends to infinity.*

THEOREM 2.3. *Let $M = M(n)$ be an $n \times n$ matrix whose entries are drawn independently from the uniform distribution over $\{0, 1, \ldots, \lfloor c_n n \rfloor\}$, where c_n tends to infinity with n. Then the probability that $AP(M) \neq ATSP(M)$ tends to 1 as n tends to infinity.*

3. Proof of Theorem 2.1. We begin with some conventions and definitions.When n is understood from context we abbreviate p_n by p, $w(n)$ by w and $M(n)$ by M; also, "permutation" will mean "permutation of $\{1, 2, \ldots, n\}$."

Let H be the weighted bipartite graph with vertex set $X \cup Y$, where $X = \{x_1, x_2, \ldots, x_n\}$ and $Y = \{y_1, y_2, \ldots, y_n\}$, and with an edge of weight m_{ij} between x_i and y_j. Let G be the complete digraph on vertex set $\{1, 2, \ldots, n\}$, in which each edge $< i, j >$ has weight m_{ij}. A *cycle cover* is a subgraph of G in which each of the n vertices has in-degree 1 and out-degree 1. The AP can be stated in any of the following equivalent forms:

- find a perfect matching of minimum weight in H;
- find a cycle cover of minimum weight in G;
- find a permutation σ to minimize $\sum_{i=1}^{n} m_{i,\sigma(i)}$.

Let the indicator variable z_{ij} be 1 if $m_{ij} = 0$ and 0 otherwise. Then the z_{ij} are independent, and each z_{ij} is equal to 1 with probability p. Emulating a useful trick due to Walkup [Wal1], we

view the z_{ij} as being generated in the following way. Let h be defined by the equation $1-p=(1-h)^5$ and let z_{ij}^k, for $i=1,2,\ldots,n$, $j=1,2,\ldots,n$ and $k=1,2,3,4,5$, be independent indicator variables, each of which is equal to 1 with probability h. Let $z_{ij}=\max_{k=1}^5 z_{ij}^k$. Then the z_{ij} are independent and each is equal to 1 with probability p. For $k=1,2$, let H_k be the bipartite graph with vertex set $X\cup Y$ and an edge between x_i and y_j if and only if $z_{ij}^k=1$. For $k=3,4,5$, let G_k be the digraph with vertex set $\{1,2,\ldots,n\}$ and an edge from i to j if and only if $z_{ij}^k=1$. The edges of G_3, G_4, and G_5, respectively, will be called *out-edges*, *in-edges*, and *patch edges*. Each type of edge will play a special role in the construction of a Hamiltonian circuit of weight $AP(M)$. It will be important that the random graphs H_1 and H_2 and the random digraphs G_3, G_4, and G_5 are completely independent. Also, let $s(n)=nh(n)$; $s(n)$ is the expected degree of a vertex in H_1 or H_2, and the expected out-degree of a vertex in G_3, G_4 or G_5. Clearly, $s(n)\geq\frac{w(n)}{5}$, and thus $s(n)$ tends to infinity if $w(n)$ does.

The construction of the desired Hamiltonian circuit proceeds in the following stages:

- (Identification of "troublesome vertices.") By considering the edges of $H_1\cup H_2$ identify a set $A\subset X$ and a set $B\subset Y$. The cardinality of $A\cup B$ is small (whp). The set $A\cup B$ contains the vertices of exceptionally small degree plus certain other vertices that are likely to be incident with edges of nonzero weight in an optimal assignment. At the same time construct a matching in H which is of minimum weight, subject to the condition that it covers the vertices in $A\cup B$ and no other vertices.
- Consider the subgraph of $H_1\cup H_2$ induced by $(X\setminus A)\cup(Y\setminus B)$. This bipartite graph has a perfect matching (whp). Combining that perfect matching with the matching constructed in the previous step, obtain an optimal assignment for H in which every nonzero-weight edge is incident with a vertex in $A\cup B$.
- The optimal assignment just constructed has the properties of a random permutation.
- Using the out-edges and in-edges, attempt to convert the original optimal assignment into a permutation with no short cycles. This process succeeds (whp).
- Using the patch edges, patch the long cycles together into a single cycle, thus solving the $ATSP$. The patching process succeeds (whp).

The overall strategy of the proof is to construct an optimal assignment while keeping the in-edges, out-edges, and patch edges (except those incident with $A\cup B$) in reserve for use in converting the optimal assignment to a tour. In the following sections we describe the algorithm in greater detail and give the proofs of the main assertions.

4. Identification of the sets A and B. Consider the directed bipartite graph D with vertex set $X\cup Y$. The edges of D are those of H_1 directed from X to Y plus those of H_2 directed from Y to X. The expected out-degree of a vertex in D is $s(n)$, which we abbreviate by s. Let $d(v)$ be the out-degree of vertex v, let $N(v)$ be the set of out-neighbors of v, and, for any set of vertices S, let $N(S)$ be the set of vertices adjacent from vertices in S.

We give an iterative construction for identifying a small set $A\cup B$ of vertices that are likely to be incident with edges of nonzero weight in an optimal assignment.

Let $W_{-1}=\{x\in X\cup Y|d(x)\leq s/2\}$. Let F_0 be a minimum weight matching in H which covers the vertices of W_{-1}. Let W_0 denote the set of vertices covered by F_0. Define a maximal sequence $(W_0,F_0),(W_1,F_1),\ldots,(W_r,F_r)=(W,F)$ where (W_i,F_i) is obtained from (W_{i-1},F_{i-1}) as follows: suppose there exists $x\notin W_{i-1}$ such that $|N(x)\cap W_{i-1}|\geq s/4$. F_i is then a minimum weight matching in H which covers W_{i-1} and x and, necessarily, one other vertex y. We then take $W_i=W_{i-1}\cup\{x,y\}$. (F_i is obtained by constructing a least cost augmenting path from x w.r.t. F_{i-1}.)

LEMMA 4.1. $|W|<3ne^{-\frac{s}{5}}$ *(whp)*.

Proof. The proof follows from two simple claims. They can easily be justified by the first moment method; the calculations are omitted.

CLAIM 1. $|W_{-1}| \leq ne^{-s/5}$ *(whp).*

CLAIM 2. $S \subseteq X \cup Y$, $|S| \leq 3ne^{-s/5}$ *implies that (whp), in* $H_1 \cup H_2$, S *contains fewer than* $2|S|$ *edges.*

Assume the conditions of the above two claims. Then $|W_0| \leq 2|W_{-1}| \leq 2ne^{-s/5}$. Now $|W_i| = |W_0| + 2i$ and W_i contains at least $is/4$ edges. If $|W| \geq 3ne^{-s/5}$ then r, the number of pairs of vertices adjoined to W_0 in constructing the set W, is greater than or equal to $r_0 = \lfloor ne^{-s/5}/2 \rfloor$. But then W_{r_0} has at most $3ne^{-s/5}$ vertices and contains at least $r_0 s/4$ edges, contradicting Claim 2. □

We then take $A = W \cap X$ and $B = W \cap Y$. The subprocess of constructing A involves counting the edges directed into $Y \setminus B$ from each vertex $x \in X \setminus A$, but does not depend at all on which particular vertices in $Y \setminus B$ are adjacent to x. Thus, $N(x) \cap (Y \setminus B)$ is of cardinality at least $s/4$ and it is a random set in the sense that the probability that it is equal to a given subset of $Y \setminus B$ depends only on the cardinality of that subset; moreover, there is no dependency among the distinct sets $N(x) \cap (Y \setminus B)$ as x ranges over $X \setminus A$. Similar statements can be made about the sets $N(y) \cap (X \setminus A)$, for $y \in Y \setminus B$. There are also no dependencies between these two collections of sets. Furthermore, in constructing W using the augmenting path approach, we did not need to consider the cost of any edge with both its endpoints in $H - W$. Thus, each such edge is still an in, out, or patch edge with probability h.

5. Construction of an optimal assignment.

LEMMA 5.1. *The subgraph of* $H_1 \cup H_2$ *induced by* $(X \setminus A) \cup (Y \setminus B)$ *has a perfect matching (whp).*

Proof. Recall that a random k-out bipartite graph on the vertex set $X \cup Y$, where X and Y are disjoint n-element sets, is constructed by having each vertex in X choose k random neighbors in Y and each vertex in Y choose k random neighbors in X. The proof follows immediately from Walkup's result [Wal2] that a random k-out bipartite graph has a perfect matching (whp) for any $k \geq 2$. □

Thus, we can obtain an optimal assignment (whp) by combining an optimal matching covering $A \cup B$ with a perfect matching in the subgraph of $H_1 \cup H_2$ induced by $(X \setminus A) \cup (Y \setminus B)$.

6. Structure of the optimal assignment. In this section we show that, if M is a random instance of the AP, then, with suitable implementation, the construction of an optimal assignment based on Lemma 5.1 yields a random permutation. Define the *equivalence class* of a matrix M as the set of all matrices obtained by permuting the columns of M. A typical member of this equivalence class, corresponding to the permutation π, is the matrix M^π defined by $m^\pi_{i,\pi(j)} = m_{ij}$. Except for a negligible fraction of the matrices in U_n^N (namely, those with two equal columns), the equivalence class of M consists of $n!$ distinct and equiprobable matrices. Let σ be the optimal assignment for M obtained by the algorithm described above. Then $\pi\sigma$ is an optimal assignment for M^π. Moreover, the algorithm for constructing the optimal assignment can be implemented so that the following hold:

- If σ is the optimal assignment constructed for M, then $\pi\sigma$ is the optimal assignment constructed for M^π;
- A, the set of troublesome rows for M, is also the set of troublesome rows for M^π.

One way to ensure this is to permute the columns of M into lexicographic order, find the set of troublesome rows and an optimal assignment in the resulting matrix, and then permute the columns back. For any fixed σ, as π ranges over all permutations of $\{1, 2, \ldots, n\}$, $\pi\sigma$ also ranges over all permutations of $\{1, 2, \ldots, n\}$. Since all the matrices in $[M]$ are equally likely, we have established that the permutation produced by the optimal assignment algorithm described above is equally likely to be any permutation.

We note some facts about random permutations. Let σ be drawn at random from the set of permutations of $\{1, 2, \ldots, n\}$. Then the following are true:

- σ has at most $2 \ln n$ cycles (whp);
- For all k,there are fewer than $w(n)^k$ cycles of length k (whp);
- There are at most $\frac{n}{w^{1/3}}$ vertices on cycles of length at most $\frac{n}{\sqrt{w}}$ (whp).

Now let σ be the optimal assignment selected by our algorithm. Then we have the following lemma.

LEMMA 6.1. *No cycle of σ has more than one-tenth of its vertices in W (whp).*

Proof. Conditioning on the event that M lies in a particular equivalence class, σ is equally likely to be any permutation, while A is a fixed set of very small cardinality (whp). A straightforward calculation shows that no cycle has more than one-twentieth of its vertices in A (whp). Indeed, given $|A| = a \leq 3ne^{-s/5}$, the expected number of cycles containing so many members of A is at most

$$\sum_{k=1}^{n} \binom{n-a}{k-\lceil k/20 \rceil} \binom{a}{\lceil k/20 \rceil} (k-1)! n^{-k} \leq \sum_{k=1}^{n} 2^k \left(\frac{a}{n}\right)^{k/20} = o(1).$$

A similar argument applies to B. □

7. Elimination of small cycles. Call a cycle in a permutation *small* if it contains fewer than $\frac{n}{\sqrt{w}}$ vertices. We now show how the out-edges and in-edges are used to convert the original optimal assignment into an optimal assignment in which no cycle is small. Our procedure is to take each small cycle of the original optimal assignment σ in turn and try to remove it without creating any new small cycles. During its execution our algorithm will designate a vertex as *dirty* when its out-edges or in-edges have been observed, so that they may no longer be considered random. The initial set of dirty vertices is the set W defined above. A vertex that is not dirty will be called *clean*. If i is clean then, independently for each j, $< i, j >$ is an out-edge with probability $\frac{s}{n}$ and $< j, i >$ is an in-edge with probability $\frac{s}{n}$. Throughout the computation, we will maintain the property that at least nine-tenths of the vertices in any remaining short cycle are clean.

We now describe the *rotation-closure algorithm* that is used to eliminate one small cycle. Let C be a small cycle of length k in the current optimal assignment. Let $\hat{k} = \min(\lceil \frac{9k}{10} \rceil, \lfloor \ln \ln n \rfloor)$. Choose $\hat{k}$ clean vertices on C. We make up to $\hat{k}$ separate attempts to remove C. The ith attempt consists of an out phase and an in phase. Let v_i be the ith of the $\hat{k}$ clean vertices selected from C, and let u_i be the predecessor of v_i on C.

7.1. The out phase. Define a *near-cycle-cover* as a digraph θ consisting of a directed path P_θ ending at a clean vertex plus a set of vertex-disjoint directed cycles covering the vertices not in P_θ. We obtain an initial near-cycle-cover by deleting edge $< u_i, v_i >$ from the current optimal assignment, thus converting the small cycle C into a path from v_i to u_i. We then attempt to obtain many near-cycle-covers by a rotation process. The state of this process is described by a rooted tree whose nodes are near-cycle-covers, with the original near-cycle-cover at the root. Consider a typical node θ consisting of a path P_θ directed from a_θ to b_θ plus a cycle cover of the remaining vertices. We obtain descendants of θ by looking at out-edges directed from b_θ. Consider an edge that is directed from b_θ to a vertex y whose predecessor x is clean. Such an edge is *successful* if either y lies on a large cycle or y lies on P_θ and the subpaths of P_θ from a_θ to x and from y to b_θ are both of length at least $\frac{n}{\sqrt{w}}$. In such cases a descendant of θ is created by deleting $< x, y >$ and inserting $< b_\theta, y >$. Once node θ has been examined, b_θ is permanently marked dirty. The tree of near-cycle-covers is grown in a breadth-first manner until the number of leaves reaches $m = \sqrt{n \ln n}$.

We shall show later that the number of vertices marked dirty throughout the entire algorithm is $o(n)$ (whp). Assuming this, noting that each path P_θ ends in a clean vertex

b_θ, and assuming that the number of vertices on short cycles is less than $\frac{n}{w^{1/3}}$ (this is true (whp)) the number of descendants of node θ is a random variable whose distribution is $BINOMIAL(n - o(n), s/n)$, and the random variables associated with distinct nodes are independent. Suppose that level t of the rooted tree describing the out phase has a vertices. Then, applying a Chernoff bound on the tails of the binomial distribution, the number of nodes at level $t+1$ lies between $\frac{as}{2}$ and $2as$, with probability greater than or equal to $1 - e^{-\frac{as}{10}}$. Hence the probability that the out phase fails to produce m leaves is (quite conservatively) at most

$$\sum_{k=1}^{\infty} e^{-ks/10} \leq e^{-s/20}.$$

7.2. The in phase. The tree produced by an out phase has m terminal nodes. Each of these is a near-cycle-cover in which the directed path begins at v_i. Let the jth terminal node be denoted G_j, and let the directed path in G_j run from v_i to x_j. During the in phase we grow rooted trees independently from all the G_j, $j = 1, 2, \ldots, m$. The process is like the out phase except that, in computing the descendants of a node θ, we fan backward along in-edges rather than forward along out-edges. For example, if a node θ with a path P_θ from a_θ to x_j is encountered, then we look for in-edges of the form $< x, a_\theta >$ such that x does not lie on a short cycle and y, the successor of x in G_θ, is clean. We then create a descendant by deleting $< x, y >$ and inserting $< x, a_\theta >$, provided that this substitution does not create a path or cycle of length less than $\frac{n}{\sqrt{w}}$. Once the descendants of θ have been computed, the node a_θ is permanently marked dirty.

Suppose that $S_1, S_2, \ldots, S_m$ are the near-cycle-covers produced by the out phase. We describe a two-stage process for producing the near-cycle-covers of the in phase which is equivalent to that described in the previous paragraph. In the first stage, imagine that we grow the trees ensuring only that edges from clean vertices are used and that y is not on a small cycle. Thus we allow new small cycles to be produced. Each tree is grown to a depth ℓ where $(w/2)^\ell \approx m$. The following is true (whp): for each tree, and each depth $t < \ell$, the ratio between the number of nodes of depth $t+1$ and the number of nodes of depth t lies between $w/2$ and $2w$. The parameter ℓ is chosen so that even if each nonleaf has only $w/2$ descendants, the tree will still have at least m leaves. Let Σ_j denote the set of initial vertices of the paths of the near-cycle-covers created from S_j in this way. We show that $\Sigma_1 = \Sigma_2 = \cdots = \Sigma_m$. In fact let $\Sigma_j^{(t)}$ denote the set of start vertices of the paths at level t in the jth tree. Clearly $\Sigma_j^{(0)} = \{v_i\}$ for all j and so assume inductively that we have $\Sigma_j^{(t)} = \Sigma_1^{(t)}$ for some $t \geq 0$. But then the clean vertices $\Sigma_j^{(t+1)}$ whose in-edges are directed into $\Sigma_j^{(t)}$ are the same as the clean vertices $\Sigma_1^{(t+1)}$ whose in-edges are directed into $\Sigma_1^{(t)}$. This completes the inductive step. We see also that from this construction the number of vertices marked dirty by this stage is at most $(2w)^\ell$ and then it is easy to see that the total number of dirty vertices produced is $O(n^{.5+o(1)})$. It follows from our analysis of the out phase that if $\mathcal{E}$ denotes the event "we fail to produce m trees in the first stage," then

$$Pr(\mathcal{E}) \leq e^{-w/20}. \tag{1}$$

We do not have to multiply the right-hand side of (1) by m because the construction above succeeds for all S_j if and only if it succeeds for S_1. In the second stage we prune the m trees we have produced by deleting any edge which involved the construction of a small cycle. For $j = 1, 2, \ldots, m$, let T_j be the pruned tree grown from root G_j during the two stages. Thus T_j is what we would get from G_j if we had followed the procedure described in the second paragraph of this section.

Let us call T_j *good* if it has m leaves and *bad* otherwise. Since the number of vertices marked dirty during the entire computation is $o(n)$, by the same analysis that was applied to the out phase, the probability that an individual T_j is bad is less than or equal to $e^{-\frac{w}{20}}$. Thus the expected number of bad trees is at most $me^{-\frac{w}{20}}$ and, by Markov's inequality, the probability that the number of bad trees is at least $m/2$ is bounded above by $2e^{-\frac{w}{20}}$. Thus,

$$Pr\left(\exists \geq \frac{m}{2} \text{ good trees}\right) \geq Pr(\mathcal{E}) - Pr\left(\exists \geq \frac{m}{2} \text{ bad trees}\right)$$
$$\geq 1 - 3e^{-w/20}.$$

Assuming that the out phase succeeds in creating a rooted tree with m leaves, and that at least half of the m trees T_j created during the in phase are good, we now have at least $m/2$ sets, each consisting of m near-cycle-covers. In the set associated with T_j, each near-cycle-cover consists of a path ending at x_j, together with a cycle cover of the remaining vertices. The m paths have distinct starting points. Since the out-edges from x_j are unconditioned, the probability that none of the $\frac{m^2}{2}$ paths is closed by an out-edge is bounded above by $(1-\frac{s}{n})^{m^2/2} \leq n^{-\frac{s}{2}}$. Thus, with probability at least $1 - n^{-\frac{s}{2}}$, one of these paths can be closed with an out-edge, and doing so creates an optimal assignment with one short cycle less than the optimal assignment that existed at the beginning of the out phase.

Thus the probability that the tth attempt at removing C fails given that the first $t-1$ attempts have also failed can be bounded by, say, e^{-Aw} for some absolute constant $A > 0$. Since we make $\hat{k}$ attempts, the probability that we fail to remove all short cycles is at most

$$\sum_{k=1}^{\lfloor \ln \ln n \rfloor} w^k e^{-\hat{k}Aw} + 2 \ln n e^{-Aw \ln \ln n} = o(1).$$

8. The patching process. At the start of the patching process we have an optimal assignment without small cycles, and the patch edges are unobserved, and thus unconditioned, except for those incident with the set of vertices $W = \{i | x_i \in A\} \cup \{j | y_j \in B\}$. Suppose we start with cycles $C_1, C_2, \ldots, C_r$, each of which contains at least $\frac{n}{\sqrt{w}}$ vertices. We describe a procedure that attempts to patch these cycles together to form a tour. The basic operation of patching together two cycles C and C' is as follows. Suppose cycle C contains an edge $< a_1, a_2 >$ and cycle C' contains an edge $< b_1, b_2 >$. If $< a_1, b_2 >$ and $< b_1, a_2 >$ are both patch edges then we can combine C and C' into a single cycle by deleting $< a_1, a_2 >$ and $< b_1, b_2 >$ and inserting $< a_1, b_2 >$ and $< b_1, a_2 >$.

We attempt to create a tour by repeatedly patching cycles together in this way. We describe a generic step in which, having patched $C_1, C_2, \ldots, C_{s-1}$ together to form a cycle C, we try to patch C_s and C together. There are at most $3ne^{-\frac{s}{5}}$ vertices in W on $C_s \cup C$. Independently, for each pair consisting of an unconditioned vertex on C_s and an unconditioned vertex on C, a patch edge is present with probability s/n. Thus, the probability that there is no pair of edges that will patch C_s and C together is bounded above by $(1 - (\frac{s(n)}{n})^2)^{(|C|-3ne^{-s/5})(|C_s|-3ne^{-s/5})}$. This is less than or equal to $e^{-w+o(1)}$. Hence the probability that the patching process fails is bounded above by $\sqrt{w}e^{-w+o(1)} = o(1)$.

We have now completed the entire proof of Theorem 2.1.

9. The proofs of Theorems 2.2 and 2.3. In proving Theorems 2.2 and 2.3, we shall describe the generation of our matrix M, the corresponding bipartite graph H, and the directed graph G in a slightly different manner.

We first generate a random $n \times n$ matrix N where each entry is drawn uniformly from $\{0, 1, \ldots, \lfloor c_n n \rfloor\}$ and these choices are independent (note that in order to combine

Theorems 2.2 and 2.3 we insist only that c_n is either constant or goes to infinity with n). We then randomly choose a permutation Π of $\{1, \ldots, n\}$ with each permutation equally likely. We obtain M by setting $m_{i\pi(j)} = n_{ij}$ (where n_{ij} is the entry in the ith row and jth column of N). Proceeding in this fashion rather than choosing the elements of M directly makes certain assertions about the independence of events obvious. We let H' be the bipartite graph which corresponds to N in the same way that H corresponds to M. We note that if we choose some optimal matching in H' then the corresponding matching in M will have the cycle cover of a random permutation.

We now need some definitions. So, consider an arbitrary matrix N and corresponding bipartite subgraph H'. By a *forced edge* of H' we mean an edge which is in every optimal matching in H'. By an *active edge* we mean an edge which is in some optimal matching of H' and has weight at least one. The first step in proving Theorems 2.2 and 2.3 is to note that if a particular weighting has a lot of forced edges then probably it will not satisfy $AP(M) = ATSP(M)$. In particular if all the edges are forced then the probability that $ATSP(M) = AP(M)$ is $\frac{1}{n}$.

The precise result we will need is that if for some weighting the corresponding H' has s forced edges then the probability that the corresponding cycle cover has a non-Hamiltonian cycle made up only of forced edges — a *forced cycle* — is the same as it would be if we took a random cycle cover and then chose s edges at random and called them forced. This follows from the manner in which we generate M. It is convenient now to give a lower bound for the probability $\pi_{t,n}$ that a random cycle cover has a cycle of length at most t (more precise esitmates are available, see for example Bollobás [B]). We will use $\pi_{t,n} \geq 1 - \frac{1}{t+1}$. To see this use induction on

$$\pi_{t,n} = \left(2t - 1 + \sum_{i=t+1}^{n-t-1} \pi_{t,i}\right)/n,$$

which is a consequence of the fact that the size of the cycle containing 1 is uniformly distributed. The following lemma then follows easily.

LEMMA 9.1. *For any weighting of N such that H' has s forced edges, the probability that the corresponding weighting of M has a forced cycle of size at most $n-1$ and hence $AP(M) \neq ATSP(M)$ is at least*

$$\max\left\{\left(1 - \frac{1}{t+1}\right)\left(\frac{s(s-1)\ldots(s-t+1)}{n(n-1)\ldots(n-t+1)}\right) \mid t = 1, 2, \ldots, n-1\right\}.$$

The second step in the proof is to note that most weightings will have many active edges because many vertices of G will not be the tail of any arc of cost zero. In fact since the probability that x is such a vertex is $(1 - \frac{1}{\lfloor c_n n \rfloor})^n$, we obtain the following lemma.

LEMMA 9.2. *The expected number of active edges for a random weighting is at least*

$$(1 + o(1))ne^{-1/c_n}.$$

The key to the proof is the following lemma which links these two results.

LEMMA 9.3. *The expected number of forced edges in a random weighting is at least the expected number of active edges.*

Combining Lemmas 9.2 and 9.3, we obtain that the expected number of forced edges is at least $(1 + o(1))ne^{-1/c_n}$. Theorem 2.2 then follows immediately from Lemma 9.1, on taking $t = 2$, i.e.,

$$\begin{aligned} Pr(AP(M) \neq ATSP(M)) &\geq Pr \text{ (forced cycle of length at most 2)} \\ &\geq E\left(\frac{2s(s-1)}{3n(n-1)}\right) \\ &\geq \frac{2E(s)(E(s)-1)}{3n(n-1)} \qquad \text{by Jensen's inequality} \\ &= (1-o(1))\frac{2e^{-2/c}}{3}. \end{aligned}$$

Again combining Lemmas 9.1, 9.2, and 9.3 we see that if c_n tends to infinity with n then Theorem 2.3 follows from

$$\begin{aligned} Pr(AP(M) \neq ATSP(M)) &\geq Pr \text{ (forced cycle of length at most } t) \\ &\geq \left(1-\frac{1}{t+1}\right)\left(1-\frac{2}{c_n}\right)^t\left(1-O\left(\frac{t^2}{n}\right)\right) \\ &= 1-o(1) \end{aligned}$$

if $t \to \infty, t = o(c_n + \sqrt{n})$.

One can tighten Theorem 2.2 slightly by insisting that the solution to $AP(M)$ contains no 1-cycles. Thus let $D(M)$ denote problem $AP(M)$ with the added constraint that the permutation should contain no 1-cycles, i.e., be a *derangement*. If the solution to $AP(M)$ is a derangement then it also solves $D(M)$ and the probability of this tends to e^{-1}. Since forced edges occur independently of the cycle structure we can see that

$$Pr(D(M) \neq ATSP(M)) \geq (1-o(1))\left(\frac{2}{3}-(1-e^{-1})\right)e^{-2/c}.$$

The question of whether or not Theorem 2.3 can be similarly strengthened remains open. The answer is almost certainly yes, but how do we prove it?

It remains only to prove Lemma 9.3.

To prove Lemma 9.3, we give an injective mapping from the (weighting, active-edge) pairs to the (weighting, forced-edge) pairs. This implies the result. Indeed, let $m = \lfloor c_n \rfloor + 1$ and Ω_e (respectively, Ω'_e) denote the set of weightings in which e is an active (respectively, forced) edge. Then

$$\begin{aligned} E(s) &= m^{-n}\sum_e |\Omega_e| \\ &\geq m^{-n}\sum_e |\Omega'_e| \qquad \text{as will be shown} \\ &= E \text{ (number of active edges)}. \end{aligned}$$

It only remains to show that $|\Omega_e| \geq |\Omega'_e|$ for all edges e. Now, given an active edge e in a weighting W, we obtain a new weighting W' by reducing the weight on e by 1 and leaving all other weights the same. We note that the cost of an optimal matching with respect to W' is one less than the cost of an optimal matching with respect to W and any optimal matching with respect to W' must use e. In our mapping, we map (W, e) to (W', e). Clearly, this gives the desired injection. This completes the proof of Lemma 9.3 and the two theorems. We note that our injection is almost a bijection because adding one to a forced edge yields an active edge in a new matching unless the forced edge has weight $\lfloor c_n n \rfloor$, which is a rare occurrence.

REFERENCES

[BaTo] E. BALAS AND P. TOTH, *Branch and bound methods*, in The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization, E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, eds., John Wiley, New York, 1985.

[B] B. BOLLOBÁS, *Random Graphs*, Academic Press, London, 1985.

[DF] M. E. DYER AND A. M. FRIEZE, *On patching algorithms for random asymmetric travelling salesman problems*, Math. Program., 46 (1990), pp. 361–378.

[Fr] A. M. FRIEZE, *An algorithm for finding hamilton cycles in random digraphs*, J. Algorithms, 9 (1988), pp. 181–204.

[K] R. M. KARP, *A patching algorithm for the non-symmetric traveling salesman problem*, SIAM J. Comput., 8 (1979), pp. 561–573.

[KS] R. M. KARP AND J. M. STEELE, *Probabilistic analysis of heuristics*, in The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization, E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, eds., John Wiley, New York, 1985.

[MiPe] D. L. MILLER AND J. F. PEKNY, *Exact solution of large asymmetric traveling salesman problems*, Science, 251 (1991), pp. 754–762.

[Wal1] D. W. WALKUP, *On the expected value of a random assignment problem*, SIAM J. Comput., 8 (1979), pp. 440–445.

[Wal2] D. W. WALKUP, *Matchings in random regular bipartite graphs*, Discrete Math., 31 (1980), pp. 59–64.

SIAM J. COMPUT.
Vol. 24, No. 3, pp. 494–504, June 1995

SCALING ALGORITHMS FOR THE SHORTEST PATHS PROBLEM*

ANDREW V. GOLDBERG†

Abstract. We describe a new method for designing scaling algorithms for the single-source shortest paths problem and use this method to obtain an $O(\sqrt{n}m \log N)$ algorithm for the problem. (Here n and m are the number of nodes and arcs in the input network and N is essentially the absolute value of the most negative arc length; arc lengths are assumed to be integral.) This improves previous bounds for the problem. The method extends to related problems.

Key words. shortest paths problem, graph theory, networks, scaling

AMS subject classifications. 68Q20, 68Q25, 68R10, 05C70

1. Introduction. In this paper we study the shortest paths problem where arc lengths can be both positive and negative. This is a fundamental combinatorial optimization problem that often comes up in applications and as a subproblem in algorithms for many network problems. We assume that the length function is integral, as is the case in most applications.

We describe a framework for designing scaling algorithms for the shortest paths problem and derive several algorithms within this framework. Our fastest algorithm runs in $O(\sqrt{n}m \log N)$ time, where n and m are the number of nodes and arcs of the input network, respectively, and the arc costs are at least $-N$.[1] Our approach is related to the cost-scaling approach to the minimum-cost flow problem [2], [14], [18], [21].

Previously known algorithms for the problem are as follows. The classical Bellman–Ford algorithm [1], [8] runs in $O(nm)$ time. Our bound is better than this bound for $N = o(2^{\sqrt{n}})$. Scaling algorithms of Gabow [12] and Gabow and Tarjan [13] are dominated by an assignment subroutine. The former algorithm runs in $O(n^{3/4}m \log N)$ time; the latter algorithm runs in $O(\sqrt{n}m \log(nN))$ time.[2] Our bound dominates these bounds. The fastest shortest paths algorithm currently known for planar graphs [9], [19] runs in $O(n^{1.5})$ time. Our algorithm runs in $O(n^{1.5} \log N)$ time on planar graphs and is competitive for small values of N.

Our framework is very flexible. In §§8 and 9 we describe two variations of the $O(\sqrt{n}m \log N)$ algorithm. The first variation seems more practical and the second variation shows the relationship between our method and Dijkstra's shortest path algorithm [6]. The flexibility of our method may lead to better running time bounds.

The shortest paths problem is closely related to other problems, such as the minimum-cost flow, assignment, and minimum-mean length cycle problems. Our method for the shortest paths problem extends to these problems. In §10 we sketch extensions to the minimum-cost flow and assignment problems. McCormick [20] shows an extension to the minimum-mean cycle problem. The resulting algorithms achieve bounds that are competitive with those of the fastest known algorithms, but are somewhat simpler.

2. Preliminaries. The input to the single-source shortest paths problem is (G, s, l), where $G = (V, E)$ is a directed graph, $l : E \to \mathbf{R}$ is a length function, and $s \in V$ is the

*Received by the editors May 28, 1992; accepted for publication (in revised form) November 22, 1993.

†Computer Science Department, Stanford University, Stanford, California 94305. This research was supported in part by the Office of Naval Research Young Investigator award N00014-91-J-1855; National Science Foundation Presidential Young Investigator grant CCR-8858097 with matching funds from AT&T, DEC, and 3M; Powell Foundation grant; and a Mitsubishi Electric Laboratories grant. Part of this work was done while the author was visiting IBM Almaden Research Center and supported by Office of Naval Research contract N00014-91-C-0026.

[1]We assume that $N \geq 2$ so that $\log N > 0$.

[2]In [12], [13] these bounds are stated in terms of C, the maximum absolute value of arc costs. As noted by an anonymous referee, it is easy to see that C can be replaced by N.

source node (see, e.g., [4], [23]). The goal is to find shortest paths distances from s to all other nodes of G or to find a negative length cycle in G. If G has a negative length cycle, we say that the problem is *infeasible*. We assume that the length function is integral. We also assume, without loss of generality, that all nodes are reachable from s in G and that G has no multiple arcs. The latter assumption allows us to refer to an arc by its endpoints without ambiguity.

We denote $|V|$ by n and $|E|$ by m. Let M be the smallest arc length. Define $N = -M$ if $M < -1$ and $N = 2$ otherwise. Note that $N \geq 2$ and $l(a) \geq -N$ for all $a \in E$.

A *price function* is a real-valued function on nodes. Given a price function p, we define a *reduced cost function* $l_p : E \to \mathbf{R}$ by

$$l_p(v, w) = l(v, w) + p(v) - p(w).$$

We say that a price function p is *feasible* if

$$l_p(a) \geq 0 \quad \forall a \in E. \tag{1}$$

For an $\epsilon \geq 0$, we say that a price function is *ϵ-feasible* if

$$l_p(a) > -\epsilon \quad \forall a \in E. \tag{2}$$

Given a price function p, we say that an arc a is *admissible* if $l_p(a) \leq 0$, and denote the set of admissible arcs by E_p. The *admissible graph* is defined by $G_p = (V, E_p)$.

If the length function is nonnegative, the shortest paths problem can be solved in $O(m + n \log n)$ time [10], or in $O(m + n \log n / \log\log n)$ time [11] in a random access machine computation model that allows certain word operations. We call such a problem *Dijkstra's shortest paths problem* [6]. Given a feasible price function p, the shortest paths problem can be solved as follows. Let d be a solution to the Dijkstra's shortest paths problem (G, s, l_p). Then the distance function d' defined by $d'(v) = d(v) + p(v) - p(s)$ is the solution to the input problem.

We restrict our attention to the problem of computing a feasible price function or finding a negative length cycle in G.

3. Successive approximation and bit scaling frameworks. Our method computes a sequence of ϵ-feasible price functions with ϵ decreasing by a factor of two at each iteration. Initially, all the prices are zero and ϵ is the smallest power of two that is greater than N. The method maintains integral prices. At each iteration, the method halves ϵ and applies the REFINE subroutine, which takes as input a (2ϵ)-feasible price function and returns an ϵ-feasible price function or discovers a negative length cycle. In the latter case, the computation halts.

LEMMA 3.1. *Suppose a price function p is integral and 1-feasible. Then for every $a \in E$, $l_p(a) \geq 0$.*

Proof. The lemma follows from the fact that $l_p(a)$ is integral and $l_p(a) > -1$. □

Bit scaling, first applied to the shortest paths problem by Gabow [12], can be used instead of successive approximation in all algorithms described in this paper. The bit scaling version of our method rounds lengths *up* to a certain precision, initially the smallest power of two that is greater than N. The lengths and prices are expressed in the units determined by the precision. Note that since the lengths are rounded up, a negative cycle with respect to the rounded lengths is also negative with respect to the input lengths.

Each iteration of the algorithm starts with a price function that is feasible with respect to the current (rounded) lengths. Note that this is true initially because of the choice of the initial unit. At the beginning of an iteration, the lengths and prices are multiplied by two, and one is subtracted from the arc lengths as appropriate to obtain the higher precision. The resulting price function is 1-feasible with respect to the current length function; the feasibility

is restored using REFINE. The method terminates when the precision unit becomes 1, which happens in $O(\log N)$ iterations. Note that the basic problem solved at each iteration of the bit scaling method is a special version of the shortest paths problem where the arc lengths are integers greater or equal to -1.

The following lemma is obvious.

LEMMA 3.2. *Both the successive approximation and the bit scaling methods terminate in* $O(\log N)$ *iterations.*

Note that if the current unit in the bit scaling method is U and the current price function is feasible with respect to the rounded length function, then the price function is U-feasible with respect to the input length function. Thus bit scaling can be viewed as a special case of successive approximation. The work on the minimum-cost flow problem [18] shows that successive approximation is more general than bit scaling; in particular, the former can be easily used to obtain strongly polynomial algorithms.

We describe bit scaling version in the algorithms. This allows us to avoid certain technical details and slightly simplifies the presentation. However, all algorithms can be restated in the successive approximation framework in a straightforward way.

When describing bit scaling implementations of REFINE, we denote the current rounded length function by l. We also use the following definitions. We call an arc (v, w) *improvable* if $l_p(v, w) = -1$, and we call a node w *improvable* if there is an improvable arc entering w.

4. Dealing with admissible cycles. Suppose that G_p has a cycle Γ. Since the reduced cost of a cycle is equal to the length of the cycle, $l(\Gamma) \leq 0$.

If $l(\Gamma) < 0$, or $l(\Gamma) = 0$ and there is an arc (v, w) such that $l_p(v, w) < 0$ and both v and w are on Γ, then the input problem is infeasible and the method terminates. Otherwise, we contract Γ and remove self-loops adjacent to the contracted node. A feasible price function on the contracted graph extends to a feasible price function on the original graph in a straightforward way.

Our algorithm uses an $O(m)$-time subroutine DECYCLE(G_p) that works as follows. Find strongly connected components of G_p (see, e.g., [22]); if a component contains a negative reduced cost arc, G has a negative length cycle; otherwise contract each component. (Note that the prices of nodes in each contracted component change by the same amount, so the reduced costs of arcs with both ends in the same component do not change.)

Suppose G_p is acyclic. Then G_p defines a partial order on V and on the subset of improvable nodes. This motivates the following definitions. A set of nodes S is *closed* if every node reachable in G_p from a node in S belongs to S. A set of nodes (arcs) S is a *chain* if there is a path in G_p containing every element of S.

5. Cut-relabel operation. In this section we study the CUT-RELABEL operation which is used by our method to transform a 1-feasible price function into a feasible one. The CUT-RELABEL operation takes a closed set S and decreases prices of all nodes in S by 1.[3] Note that the operation preserves integrality of the prices (and therefore integrality of the reduced costs).

LEMMA 5.1. *The* CUT-RELABEL *operation does not create any improvable arcs.*

Proof. The only arcs whose reduced cost is decreased by CUT-RELABEL are the arcs leaving S. Let a be such an arc. The relabeling decreases $l_p(a)$ by 1. Before the relabeling, S is closed and therefore $l_p(a) > 0$. By integrality, $l_p(a) \geq 1$. After the relabeling, $l_p(a) \geq 0$. □

The above lemma implies that CUT-RELABEL does not create improvable nodes. The next lemma shows how to use this operation to reduce the number of improvable nodes.

[3] Alternatively, the operation can decrease prices of all nodes of S by the maximum amount ϵ' such that Lemma 5.1 holds.

LEMMA 5.2. *Let p be a 1-feasible price function. Let S be a closed set of nodes, and let $X \subseteq S$ be a set of improvable nodes such that every improvable arc entering a node of X crosses the cut defined by S. After the set S is relabeled, nodes in X are no longer improvable.*

Proof. Let p' be the price function after the relabeling. Let $w \in X$ and let (v, w) be an improvable arc with respect to p. By the statement of the lemma, $v \notin S$. Thus the relabeling increases l_p by 1, and, by 1-feasibility of p, $l_{p'}(v, x) \geq 0$. □

A simple algorithm based on CUT-RELABEL applies the following procedure to every improvable node v.

1. DECYCLE(G_p).
2. $S \leftarrow$ set of nodes reachable from $\{v\}$ in G_p.
3. CUT-RELABEL(S).

It is easy to see that given a 1-feasible price function, this algorithm computes a feasible one in $O(nm)$ time.

6. Faster algorithm. In this section we introduce an $O(\sqrt{n}m \log N)$ algorithm for finding a feasible price function. Let k denote the number of improvable nodes. At each iteration, the algorithm either finds a closed set S such that applying CUT-RELABEL to S reduces the number of improvable nodes by at least $\sqrt{k}$, or a chain S such that applying ELIMINATE-CHAIN to S reduces the number of improvable nodes by at least $\sqrt{k}$. (The ELIMINATE-CHAIN operation is described in the next section.) An iteration takes linear time and is based on the results of §§5 and 7 and the following lemma, which is related to Dilworth's theorem (see, e.g., [7]).

LEMMA 6.1. *Suppose G_p is acyclic. Then there exists a chain $S \subseteq E$ such that S contains at least $\sqrt{k}$ improvable arcs or a closed set $S \subseteq V$ such that relabeling S reduces the number of improvable nodes by at least $\sqrt{k}$. Furthermore, such an S can be found in $O(m)$ time.*

Proof. Construct a graph G' by adding a source node r to G_p and arcs from r to all nodes in V. Note that G' is acyclic. Define $l'(a) = l_p(a)$ for all $a \in E_p$ and $l'(a) = 0$ for the newly added arcs a. The absolute value of the path length with respect to l' is equal to the number of improvable arcs on the path. Let $d' : V \rightarrow \mathbf{R}$ give the shortest paths distances from r with respect to l' in G'. Since G' is acyclic, d' can be computed in linear time. Define $D = \max_V |d'|$.

If $D \geq \sqrt{k}$, then a shortest path from r to a node v with $d'(v) = -D$ contains a chain with at least $\sqrt{k}$ improvable arcs.

If $D < \sqrt{k}$, then the partitioning of the set of improvable nodes according to the value of d' on these nodes contains at most $\sqrt{k}$ nonempty subsets. Let X be a subset containing the maximum number of improvable nodes and let i be the value of d' on X. Observe that X contains at least $\sqrt{k}$ improvable nodes. Define $S = \{v \in V | d'(v) \leq i\}$.

Clearly $X \subseteq S$. Also, S is closed. This is because if $v \in S$ and there is a path from v to w in G_p, then the length of this path with respect to l' is nonpositive, so $d'(w) \leq d'(v) \leq i$ and therefore $w \in S$.

We show that after CUT-RELABEL is applied to S, nodes in X are no longer improvable. Let $x \in X$ and let (v, x) be an improvable arc. Then $l'(v, x) = -1$ and therefore $d'(v) > d'(x) = i$. Thus $v \notin S$ and (v, w) is not improvable after relabeling of S. □

The efficient implementation of REFINE is described in Fig. 1. The implementation reduces the number of improvable nodes k by at least $\sqrt{k}$ at each iteration by eliminating cycles in G_p, finding S as in Lemma 6.1, and eliminating at least $\sqrt{k}$ improvable nodes in S using techniques of §§4, 5, and 7. In §7 below we describe a linear time implementation of ELIMINATE-CHAIN. This implies that an iteration REFINE runs in linear time.

LEMMA 6.2. *The implementation of* REFINE *described in this section runs in $O(\sqrt{n}m)$ time.*

```
procedure REFINE(p);
    k ← the number of improvable nodes;
    repeat
        DECYCLE(G_p);
        S ← a chain or a set as in Lemma 6.1;
        if S is a chain then
          ELIMINATE-CHAIN(S);
        else
          CUT-RELABEL(S);
        k ← the number of improvable nodes;
    until k = 0;
    return(p);
end.
```

FIG. 1. *An efficient implementation of* REFINE.

Proof. We need to bound the number of iterations of REFINE. Each iteration reduces k by at least $\sqrt{k}$, and $O(\sqrt{k})$ iterations reduce k by at least a factor of two. The total number of iterations is bounded by

$$\sum_{i=0}^{\infty} \sqrt{\frac{n}{2^i}} = O(\sqrt{n}). \qquad \square$$

Lemmas 3.2 and 6.2 imply the following result.

THEOREM 6.3. *The shortest paths algorithm with* REFINE *implemented as described in this section runs in* $O(\sqrt{nm} \log N)$ *time.*

7. Eliminate-chain subroutine. Suppose that G_p is acyclic and let Γ be a path in G_p. Let $(v_1, w_1), \ldots, (v_t, w_t)$ be the collection of all improvable arcs on Γ such that for $1 \leq i < j \leq t$, the path visits v_j before v_i (i.e., v_1 is visited last). By definition, nodes $w_1, \ldots, w_t$ are improvable. In this section we describe a subroutine ELIMINATE-CHAIN that modifies p so that the nodes $w_1, \ldots, w_t$ are no longer improvable and no new improvable nodes are created, or finds a negative length cycle in G. The subroutine runs in $O(m)$ time.

At iteration i, ELIMINATE-CHAIN finds the set S_i of all nodes reachable from w_i in the admissible graph and applies CUT-RELABEL to S_i. If w_i is improvable after the relabeling, the algorithm concludes that the problem is infeasible.

LEMMA 7.1. *The path* Γ *is always admissible. If* w_i *is improvable after iteration* i*, then the problem is infeasible.*

Proof. The price function is modified only by CUT-RELABEL. At iteration i, S_i contains w_i, all its successors on Γ, and no other nodes of Γ (by induction on i). Therefore $l_p(v_i, w_i)$ changes exactly once during iteration i, when it increases by 1. The arc (v_i, w_i) is improvable before the change, and admissible after the change. Reduced costs of other arcs on Γ do not change during the execution of ELIMINATE-CHAIN.

Suppose w_i is improvable immediately after iteration i. Then there must be a node v such that (v, w_i) is improvable and $v \in S_i$. By construction of S_i, there must be an admissible path from w_i to v. This path together with the arc (v, w_i) forms a negative length cycle. $\square$

Lemmas 5.1 and 7.1 imply that the implementation of ELIMINATE-CHAIN is correct. Next we show how to refine this implementation to achieve $O(m)$ running time. The key fact that allows such an implementation is that the sets S_i are nested.

First, we contract the set of nodes S_i at every iteration. The reason for contracting is to allow us to change the prices of nodes in S_i efficiently (these prices change by the same

amount). The CONTRACT(S_i) operation collapses all nodes of S_i into one node s_i and assigns the price of the new node to be zero. (The price of s_i is actually an increment to the prices of the nodes in S_i.) Reduced costs of the arcs adjacent to the new node remain the same as immediately before CONTRACT. Note that we have at most one contracted node at any point during ELIMINATE-CHAIN, but contracted nodes can be nested.

The UNCONTRACT(s_i) operation, applied to a contracted node s_i, restores the graph as it was just before the corresponding CONTRACT operation and adds $p(s_i)$ to prices of all nodes in S_i. At the end of the chain elimination process, we apply UNCONTRACT until the original graph is restored.

Contraction is used for efficiency only and does not change the price function computed by ELIMINATE-CHAIN, because by Lemma 7.1 $S_i \subseteq S_j$ for $1 \leq i < j \leq t$.

Second, we implement the search for the nodes reachable from w_i's in the admissible graph in a way similar to Dial's implementation [5] of Dijkstra's algorithm.[4] Our implementation uses a priority queue that holds items with integer key values in the range $[0, \ldots, 2n]$; the amortized cost of the priority queue operations is constant. We assume the following queue operations.

- enqueue(v, Q): add a node v to a priority queue Q.
- min(Q): return the minimum key value of elements on Q.
- extract-min(Q): remove a node with the minimum key value from Q.
- decrease-key(v, x): decrease the value of key(v) to x.
- shift(Q, δ): add δ to the key values of all elements of Q.

All of these operations except shift are standard; a constant time implementation of shift is trivial.

Note that if p is 1-feasible and $l_p(a) \geq 2n$, then a can be deleted from the graph. This is because in the current iteration, the reduced cost of an arc can decrease by at most n; at the next iteration, by at most $n/2$ (measured in the current units), and so on. Thus the reduced cost of a will remain nonnegative from now on. We assume that such arcs are deleted as soon as their reduced costs become large enough.

We define the key assignment function h that maps reduced costs into integers as follows.

$$h(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{otherwise.} \end{cases}$$

During the chain elimination computation, each node is *unlabeled*, *labeled*, or *scanned*. Unlabeled nodes have infinite keys; other nodes have finite keys. The priority queue Q contains labeled nodes. Initially all nodes are unlabeled. At the beginning of iteration i, key(w_i) is set to zero and w_i is added to Q. While Q is not empty and the minimum key value of the queue nodes is zero, a node with the minimum key value is extracted from the queue and scanned as in Dijkstra's algorithm except that $h(l_p(a))$ is used instead of $l_p(a)$ (see Fig. 2). When this process stops, the scanned nodes are contracted, the new node is marked as scanned, and its key is set to zero. Then the price of the new node is decreased by 1 and shift(Q, -1) is executed. This concludes iteration i.

Next we prove correctness of the implementation.

LEMMA 7.2. *The sets S_i are computed correctly for every $i = 1, \ldots, t$.*

Proof. For convenience we define $S_0 = \emptyset$. Consider an iteration i. It is enough to show that S_i is correct if $1 \leq i \leq t$ and S_{i-1} is correct.

Let v be a node on Q with the zero key value. We claim that v is reachable from w_i in the current admissible graph. To see this, consider two cases. If v was a node on Q with zero key

[4] In §9 we show that Dial's implementation can be used directly. The implementation described in this section, however, gives a better insight into the method.

```
procedure SCAN(v);
    for all (v, w) do
        if key(w) = ∞ then
            mark w as labeled;
            key(w) ← l_p(v, w);
            insert(w, Q);
        else if w is labeled and key(w) < h(l_p(v, w)) then
            decrease-key(w, l_p(v, w));
    mark v as scanned;
end.
```

FIG. 2. *The scan operation.*

value at the beginning of the iteration, then v is reachable from w_i by Lemma 7.1. Otherwise, key of v became zero when an arc (u, v) was scanned. We can make an inductive assumption that u is reachable from w_i. By definition of h, $h(u, v) = 0$ implies that $l_p(u, v) \leq 0$, and therefore v is reachable from w_i.

Let Γ be an admissible path originating at w_i. It is easy to see by induction on the number of arcs on Γ that all nodes on Γ are scanned and added to S_i.

It follows that at the end of iteration i, S_i contains all nodes reachable from w_i in the admissible graph. □

LEMMA 7.3. ELIMINATE-CHAIN *runs in* $O(m)$ *time.*

Proof. Each node is scanned at most once because a scanned node is marked as such and never added to Q. A contracted node is never scanned. The time to scan a (noncontracted) node is proportional to degree of the node, so the total scan time is $O(m)$.

The time of a CONTRACT operation is $O(1 + n')$, where n' is the number of nodes being contracted. The number of CONTRACT operations is at most n and the sum of n' values over all CONTRACT operations is at most $2n$. Thus the total cost of contract operations is $O(n)$.

The cost of an UNCONTRACT operation is $O(1 + n')$, where n' is the same as in the corresponding CONTRACT operation. Thus the total time for these operations is $O(2n)$. □

8. Alternative chain elimination. In this section we describe an algorithm based on an alternative implementation of REFINE. We call this implementation REFINE-P. The algorithm runs in $O(\sqrt{nm} \log N)$ time.

REFINE-P works in iterations, which we call *passes*. At the beginning of every pass we check for negative cycles and eliminate zero length admissible cycles using DECYCLE. Then we compute distances d' defined in the proof of Lemma 6.1. Given a nonnegative integer M, we define the *key function*

$$\delta(v) = \min(-d'(v), M) \quad \forall v \in V.$$

(We discuss the choice of initial value of M later.) Sometimes we refer to $\delta(v)$ as the *key* of v. Let V_M denote the set of nodes with key value M. At each iteration of a pass, CUT-RELABEL is applied to V_M. Then keys of nodes in V_M and all nodes reachable from V_M in the admissible graph are changed to $M - 1$ and M is decreased by one. This process is repeated until M reaches zero; at this point the pass terminates. A pass can be implemented to run in linear time; the implementation is similar to that of ELIMINATE-CHAIN. We leave the details to the reader.

The next lemma implies that CUT-RELABEL in used correctly in a pass.

LEMMA 8.1. *Immediately before a* CUT-RELABEL *operation is applied by a pass,* V_M *is closed with respect to the current admissible graph.*

Proof. Before the first CUT-RELABEL operation, V_M is closed by of the definition of δ. The admissible graph is changed only by the CUT-RELABEL operations, and after every such operation a search is done to enforce the closeness of V_M. □

Note that the function d' is well defined if the admissible graph does not have negative cycles.

LEMMA 8.2. *If at the beginning of an iteration of a pass the admissible graph is acyclic, then*

$$\delta(v) = \min(-d'(v), M) \quad \forall v \in V.$$

Proof. The proof is by induction on the number of iterations. Keys are initialized so that the statement of the lemma holds before the first iteration. Suppose that the statement is true immediately before iteration i, and show that it holds immediately after the iteration.

The d' value of nodes in V_M increases by one, and the keys of these nodes are decreased by one at the end of the iteration. The d' values of a node outside V_M changes only if this node becomes reachable from V_M in the admissible graph, in which case the new d' value of this node is $-(M-1)$ or less. The keys of the nodes that become reachable are correctly set to $M-1$. □

Recall that $D = \max_V |d'|$.

LEMMA 8.3. *Suppose that the value of M at the beginning of a pass is equal to t such that $0 < t \leq D$, and the admissible graph does not contain negative cycles throughout the pass. Then the pass decreases the number of improvable nodes by at least t.*

Proof. Given $v, w \in V$, we say that $v \succ w$ if there is a negative reduced cost path from v to w in the admissible graph. If the admissible graph does not contain negative cycles, then "$\succ$" defines a partial order on V.

Consider the beginning of an iteration of a pass, Let v be a maximum element (with respect to "$\succ$") of the set of nodes with key value M. By the previous lemma, v is an improvable node. By the choice of v, if (u, v) is an improvable arc then $u \notin V_M$. Therefore v is no longer improvable at the end of the iteration.

Each iteration of the pass reduces the number of improvable nodes, and the number of iterations is t. □

Next we discuss the choice of initial value of M. Define d_i to be the number of improvable nodes with d' value of $-i$ (in the beginning of a pass). If the initial value of M is i, $0 < i \leq D$, and there are no negative cycles, the number of improvable nodes is reduced by at least d_i by the first application of CUT-RELABEL. Combining this observation with the above lemma, we conclude that the pass reduces the number of improvable nodes by $\max(i, d_i)$. A more careful analysis shows that the improvement is at least $i + d_i - 1$, since all improvable nodes with an initial d' value of i and at least one improvable node for each value of j, $0 < j < i$, are no longer improvable after a pass. Define $k_i = i + d_i - 1$, and set M to the index that maximizes k_i. By an argument of Lemma 6.1, $k_M = \Omega(\sqrt{n})$. This implies the following theorem.

THEOREM 8.4. *With the above choice of the initial value of M, the alternative implementation of* REFINE *runs in $O(\sqrt{n}m)$ time.*

We would like to note that in practice, a pass is likely to reduce the number of improvable nodes by more then k_i, and it may be more advantageous to chose higer initial values for M. The algorithm performance is likely to be better than the above worst-case bound suggests.

9. Chain elimination using Dijkstra's algorithm. In this section we show yet another implementation of ELIMINATE-CHAIN. This implementation uses Dial's implementation of Dijkstra's algorithm [5], and does not use the CUT-RELABEL operation explicitly.

Let Γ be a path in G_p. An auxiliary network A is defined as follows.

- Let d' be the distance function on Γ with respect to l_p from the beginning of Γ to all nodes on Γ.

- Define $l'(a) = \max(0, l_p(a))$.
- Define $d'(v) = 0$ for v not on Γ.
- Add a source node t, connect t to all $v \in V$ and define $l'(t, v) = n + d'(v)$.

ELIMINATE-CHAIN works as follows.

1. Construct the auxiliary network A.
2. Compute shortest paths distances d in A with respect to l'.
3. $\forall v \in V$, $p'(v) \leftarrow p(v) + d(v) - n$.
4. Replace p by p'.

LEMMA 9.1. *The above version of* ELIMINATE-CHAIN *can be implemented to run in linear time.*

Proof. The fact that all steps of ELIMINATE-CHAIN except for the shortest paths computation take linear time is obvious. The shortest paths computation takes linear time if Dial's implementation [5] of Dijkstra's algorithm is used. This is because l' is nonnegative and the source is connected to the other nodes by arcs of length at most n. □

LEMMA 9.2.

1. *p' is integral.*
2. $\forall a \in E$, $l_{p'} \geq -1$.
3. ELIMINATE-CHAIN *does not create improvable arcs.*

Proof. The first claim follows from the fact that l' is integral. The last two claims follow from the observation that l'_d is nonnegative and, for $a \in E$, $l'_d(a) - l_{p'}(a) = 1$ if a is improvable and 0 otherwise. □

LEMMA 9.3. *If the problem is feasible, then $\forall v$ on Γ $p'(v) = p(v) + d'(v)$.*

Proof. Clearly $p'(v) \leq p(v) + d'(v)$. Assume for contradiction that for some node v on Γ, $p'(v) < p(v) + d'(v)$. For the shortest path P in A from t to v, we have $l'(P) < n + d'(v)$ and therefore $l_p(P) < n + d'(v)$. Let (t, w) be the first arc of P, and let Q be P with (t, w) deleted. We have

$$l_p(Q) = l_p(P) - n - d'(w) < d'(v) - d'(w).$$

Note that since l' is nonnegative, w must be a successor of v on Γ. Let R be the part of Γ between v and w. By the definition of d',

$$l_p(R) = d'(w) - d'(v).$$

Thus $l_p(Q) + l_p(R) < 0$. This is a contradiction because the paths Q and R form a cycle. □

LEMMA 9.4. *If the problem is feasible and v is an improvable node on Γ with respect to p, then v is not improvable with respect to p'.*

Proof. Assume for contradiction $\exists (u, v) \in E : l_{p'}(u, v) < 0$. Let P be the shortest path in A from t to u, let (t, w) be the first arc on P, and let Q be P with (t, w) deleted. Note that $d(u) \leq d(v)$, because otherwise $l_{p'}(u, v)$ cannot be negative. Therefore w must be a successor of v on Γ. Let R be the portion of Γ between v and w.

Since Q is a shortest path, we have $l'_d(Q) = 0$. This implies $l_{p'}(Q) \leq 0$. By the previous lemma $l_{p'}(R) = 0$. Therefore the cycle formed by R, Q, and (u, v) has a negative reduced cost with respect to p'. This is a contradiction. □

Remark. Implications of Lemma 9.4 are stronger than those of Lemma 7.1: if the problem is feasible, the former lemma guarantees that all improvable nodes on Γ are "fixed," and the latter guarantees only that the nodes that are heads of the improvable arcs on Γ are "fixed."

10. Extensions to the minimum-cost circulation and assignment problems. Our shortest path method extends to the minimum-cost circulation problem. The intuitive difference is that when a shortest path algorithm finds a negative cycle, it terminates; when the corresponding minimum-cost circulation algorithm finds a negative cycle, it increases the flow around the

cycle so that an arc on the cycle becomes saturated, and continues. In our discussion below, we assume that the reader is familiar with [17], [18]. We denote the reduced costs by c_p and the residual graph by G_f.

We define admissible arcs to be residual arcs with negative reduced costs, as in [17], [18]. Without loss of generality, we assume that a feasible initial circulation is available. A simple algorithm based on the CUT-RELABEL operation does the following at each iteration. First, it cancels admissible cycles; this can be done in $O(m \log n)$ time (see, e.g., [17]). Next, the algorithm picks an improvable node v, finds the set S of nodes reachable from v in the admissible graph, and executes CUT-RELABEL(S). The resulting algorithm runs in $O(nm \log n \log(nC))$ time (note that the initial flow may have residual arcs with reduced cost of $-C$ with respect to the zero price function). We can also use the TIGHTEN operation to obtain a minimum-cost flow algorithm with the same running time. These algorithms are variations of the tighten-and-cancel algorithms of [17].

In the above minimum-cost flow algorithms, the admissible graph changes due to flow augmentations in addition to price changes. Because of this fact, our analysis of the improved algorithms for the shortest paths problem does not seem to extend to the minimum-cost flow problem. In the special case of the assignment problem, the analysis of the improved shortest path algorithm can be extended to obtain an $O(\sqrt{n}m \log(nC))$ time algorithm. This bound matches the fastest known scaling bound [13], but the algorithm is different. The idea is to define the admissible graph and improvable arcs so that an improvable node has exactly one improvable arc going into it and the residual capacity of this arc is one. This is possible because of the special structure of the assignment problem. When an admissible cycle is canceled, all improvable arcs on this cycle are saturated and there are no improvable nodes on the cycle after the cancellation.

11. Concluding remarks. We described a framework for designing scaling algorithms. The CUT-RELABEL operation can be used to design algorithms within this framework. The framework is very flexible and can be used to design numerous algorithms for the problem. Using these results, we improved the time bound for the problem. We believe that further investigation of this framework is a promising research direction.

One can apply the version of ELIMINATE-CHAIN described in §9 without using scaling. It can be shown that in this case if the problem is feasible, all negative reduced costs of arcs on Γ are changed to nonnegative ones, and reduced costs of other arcs do not become more negative. This suggests a possibility of solving the general shortest paths problem in $O(\sqrt{m})$ Dijkstra shortest paths computations. The problem, however, is that our way of dealing with the first case of Lemma 6.1 does not work without scaling.

Our definition of ϵ-feasibility corresponds to that of ϵ-optimality for minimum cost flows [14], [18]. If one follows [14], [18] faithfully, however, one would define ϵ-feasibility using $l_p(a) \geq -\epsilon$ instead of (2) and not consider arcs with zero reduced costs admissible. Under these definitions, the admissible graph cannot have zero length cycles, so there is no need for DECYCLE. However, these definitions seem to lead to an $O(\log(nN))$ bound on the number of iterations of the scaling loop of the method. The *tighten* operation described in [17] also leads to an implementation of the method that runs in $O(\log(nN))$ iterations of the scaling loop.

The techniques introduced in this paper have a practical impact. In particular, the techniques of §8 proved to be crucial in our implementation of price update computation in a minimum-cost flow algorithm [15], which resulted in a significant improvement of performance.

Preliminary experiments with the algorithm of this paper, conducted as a part of the experimental study described in [3], suggest that the algorithm is not the best one to use in

practice. Although on some problem families the algorithm significantly outperformed the classical methods, it was dominated by the algorithm of [16] on all problem classes studied.

The algorithms we discussed scale ϵ by a factor of two. Any factor greater than one can be used instead without affecting the asymptotic time bounds. The method can be modified to maintain a tentative shortest path tree. When the algorithm terminates, this tree is the shortest path tree. This eliminates the need for the Dijkstra computation at the end of the algorithm.

Acknowledgments. I am grateful to Tomasz Radzik for suggesting an important idea for the proof of Lemma 6.2 and the stronger statement of Lemma 9.4, and to Bob Tarjan for suggesting a clean implementation of DECYCLE. I would also like to thank Serge Plotkin, Éva Tardos, and David Shmoys for useful discussions and comments on a draft of this paper.

REFERENCES

[1] R. E. BELLMAN, *On a routing problem*, Quart. Appl. Math., 16 (1958), pp. 87–90.

[2] R. G. BLAND AND D. L. JENSEN, *On the computational behavior of a polynomial-time network flow algorithm*, Math. Programming, 54 (1992), pp. 1–41.

[3] B. V. CHERKASSKY, A. V. GOLDBERG, AND T. RADZIK, *Shortest Paths Algorithms: Theory and Experimental Evaluation*, Technical report STAN-CS-93-1480, Department of Computer Science, Stanford University, Stanford, CA, 1993.

[4] T. H. CORMEN, C. E. LEISERSON, AND R. L. RIVEST, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[5] R. B. DIAL, *Algorithm 360: Shortest path forest with topological ordering*, Comm. ACM, 12 (1969), pp. 632–633.

[6] E. W. DIJKSTRA, *A note on two problems in connection with graphs*, Numer. Math., 1 (1959), pp. 269–271.

[7] R. P. DILWORTH, *A decomposition theorem for partially ordered sets*, Ann. Math., 51 (1950), pp. 161–166.

[8] L. R. FORD, JR. AND D. R. FULKERSON, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

[9] G. FREDERICKSON, *Fast algorithms for shortest paths in planar graphs, with applications*, SIAM J. Comput., 16 (1987), pp. 1004–1022.

[10] M. L. FREDMAN AND R. E. TARJAN, *Fibonacci heaps and their uses in improved network optimization algorithms*, J. Assoc. Comput. Mach., 34 (1987), pp. 596–615.

[11] M. L. FREDMAN AND D. E. WILLARD, *Trans-dichotomous algorithms for minimum spanning trees and shortest paths*, in Proc. 31st IEEE Annual Symposium on Foundations of Computer Science, 1990, pp. 719–725.

[12] H. N. GABOW, *Scaling algorithms for network problems*, J. Comput. Systems Sci., 31 (1985), pp. 148–168.

[13] H. N. GABOW AND R. E. TARJAN, *Faster scaling algorithms for network problems*, SIAM J. Comput., 18 (1989), pp. 1013–1036.

[14] A. V. GOLDBERG, *Efficient Graph Algorithms for Sequential and Parallel Computers*, Ph.D. Thesis, M.I.T., Cambridge, MA, January 1987 (Also available as Technical report TR-374, Lab. for Computer Science, M.I.T., 1987).

[15] ———, *An efficient implementation of a scaling minimum-cost flow algorithm*, in Proc. 3rd Integer Prog. and Combinatorial Opt. Conf., 1993, pp. 251–266.

[16] A. V. GOLDBERG AND T. RADZIK, *A heuristic improvement of the Bellman-Ford algorithm*, Appl. Math. Lett., 6 (1993), pp. 3–6.

[17] A. V. GOLDBERG AND R. E. TARJAN, *Finding minimum-cost circulations by canceling negative cycles*, J. Assoc. Comput. Mach., 36 (1989), pp. 873–886.

[18] ———, *Finding minimum-cost circulations by successive approximation*, Math. Oper. Res., 15 (1990), pp. 430–466.

[19] R. J. LIPTON AND R. E. TARJAN, *A separator theorem for planar graphs*, SIAM J. Appl. Math., 36 (1979), pp. 177–189.

[20] S. T. MCCORMICK, *Approximate binary search algorithms for mean cuts and cycles*, Oper. Res. Lett., 14 (1993), pp. 129–132.

[21] H. RÖCK, *Scaling techniques for minimal cost network flows*, in U. Pape, ed., Discrete Structures and Algorithms, Carl Hansen, Münich, 1980, pp. 181–191.

[22] R. E. TARJAN, *Depth-first search and linear graph algorithms*, SIAM J. Comput., 1 (1972), pp. 146–160.

[23] ———, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

SIAM J. COMPUT.
Vol. 24, No. 3, pp. 505–510, June 1995

007

COUNTEREXAMPLES FOR DIRECTED AND NODE CAPACITATED CUT-TREES*

ANDRÁS A. BENCZÚR†

Abstract. We show that there is no cut-tree for various connectivity concepts, hence pointing out errors in the papers of Schnorr [*SIAM J. Comput.*, 8 (1979), pp. 265–275] and Gusfield and Naor [*Networks*, 21 (1991), pp. 505–520]. Gomory and Hu [*SIAM J. Appl. Math.*, 9 (1961), pp. 551–560] constructed a cut-tree for undirected graphs which compactly represents a minimum cut for each pair of vertices. This has a straightforward generalization to directed Eulerian graphs, cf. Gupta [*SIAM J. Appl. Math.*, 15 (1967), pp. 168–171]. A generalization for arbitrary directed graphs was given by Schnorr. There is a well-known transformation of vertex connectivity to directed edge connectivity; directed edge cuts correspond to vertex cuts in some weak sense. The result of Schnorr was later applied by Gusfield and Naor for such a cut-tree construction. In this paper counterexamples are described to show that for directed graphs there is no cut-tree and therefore the cut-tree results of Schnorr and Gusfield and Naor are incorrect. Our final example shows that, without weakening the notion of vertex connectivity, it is impossible to construct vertex cut-trees for undirected graphs in general.

Key words. Gomory–Hu trees, minimum cuts, graph connectivity

AMS subject classifications. 05C40, 90B10

1. Introduction.

A theorem of Gomory and Hu [5] states that the minimum edge cuts between all pairs of vertices in an undirected edge capacitated graph G can be represented by a tree as follows. First, the value of a minimum cut between $s, t \in G$ is equal to the smallest of the capacities of the edges on the unique s–t path in the tree. Second, the removal of any edge of capacity c from the tree separates the vertices of the tree into two classes, in which the cut in G given by this partition has capacity c as well.

Trees satisfying both conditions are called *cut-equivalent trees* or, in short, *cut-trees*; when only the first condition is satisfied we call them *flow-equivalent trees*. Note that both flow-equivalent and cut-trees can be computed by $n - 1$ max-flow computations [5] and they encode all the $\binom{n}{2}$ minimum cut values. By looking at a cut-tree we can also find one such minimum cut for any pair of nodes.

It is important to note that while a cut-equivalent tree is always flow equivalent by definition, the converse does not hold. In general, the number of different flow-equivalent trees corresponding to a graph is considerably more than that of the cut-trees; for example, it is true that each graph has a flow-equivalent tree which is a path (see [4]).

As a generalization of the Gomory–Hu theorem, Cheng and Hu [2] proved the existence of flow (*but not cut*) equivalent trees for the case when an *arbitrary* value is assigned to the (undirected) cuts of the graph. The resulting tree represents all the minimum cuts of pairs of vertices with respect to this weight.

A natural question arises: can we construct equivalent trees for directed graphs? As a simple instance, the problem for directed Euler graphs easily reduces to undirected graphs and hence a cut-tree exists (cf. Gupta [7]). In general, however, we cannot ask for a compact representation of all the $n(n-1)$ directed cuts, since there can be $(n+2)(n-1)/2$ different values among them; see [3].

Not everything is lost, however. For a given pair of vertices, there are two different maximum flow values depending on which direction we choose. Let us consider the smaller

*Received by the editors August 31, 1993; accepted for publication (in revised form) November 24, 1993.

†Department of Mathematics, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. This research was done while the author was a student at the Mathematical Institute, Hungarian Academy of Sciences, and the Department of Computer Science, Eötvös University, Hungary. This research was partially supported by Hungarian National Foundation for Scientific Research (OTKA) grants 1909, 7559, and 2118.

one only. For each cut, the smaller of the capacities of the cut itself and the cut directed the other way are assigned. Then this assignment is symmetric, i.e., the values correspond to the *undirected* cuts of the graph. Hence the result of Cheng and Hu [2] can be applied to this symmetric directed case to construct flow-equivalent trees.

What about the existence of cut-trees for the above symmetric directed case? Schnorr presented an algorithm for generating such cut-trees in [10]. One aim of this paper is to show that Schnorr's result was *incorrect*, since there is no cut-tree for this case in general. (We note here that Schnorr also presented a flow-equivalent tree algorithm for this case and that is correct, but it is less efficient than the Cheng–Hu algorithm.)

Another possible task is to find a compact representation of minimum vertex cuts or cut values. One possible way to deal with vertex cuts is to transform the original graph to a directed graph (see §3). There a pair of vertices corresponds to each vertex of the original graph. Note that the connectivity properties of the transformed graph are different from the original vertex connectivities. The two inconvenient differences are the following. First, it is possible that the minimum u–v cut is u itself; this means that in the original graph u and v are separated by removing u. Second, we can get no information about vertex connectivity between vertices connected by an edge. Furthermore, while a vertex cut can have an arbitrary large number of disconnected components, a corresponding edge cut has always two components. (Consequently there may be several corresponding edge cuts.) To emphasize the difference of this notion from that of the vertex cuts, we call the former *separations* (for precise definition see §3).

The result of Cheng and Hu can be applied then to construct a flow-equivalent tree for *separations* in the transformed directed graphs (as noted in [8]). Following this approach, we may ask if cut-trees for these directed graphs exist. Applying the incorrect result of Schnorr [10], Gusfield and Naor [8] immediately got a cut-tree construction for minimum separations (on the duplicated vertex set). The existence of such a cut-tree will also be disproved in this paper.

Even for separations, it is probably more natural to construct equivalent trees with vertices from the original (and not the duplicated) vertex set of the graph. Granot and Hassin [6] have such a construction for (in our terms) undirected flow-equivalent trees and Benczúr [1] has one for cut-trees. There, as for the Gomory–Hu trees, the minimum separation value is the least of the capacities among edges of the tree on the path connecting the vertices investigated.

In contrast to separations, let us consider vertex cuts. Using the Cheng–Hu [2] result we can construct flow-equivalent trees by transforming the vertex-capacitated graph to a directed edge-capacitated one and assigning infinite cut-values to the trivial single-point cuts. Conversely, we show an example in which we cannot represent minimum vertex cuts by a cut-tree. (There are a lot of hard minimum vertex cut structures; for example, two K_n-s joined by n edges forming a matching is n-connected and has 2^n minimum capacity cuts.)

Finally, we note that the paper of Benczúr [1] is a counterpart to this paper; it proposes a graph connectivity model (separations in mixed edge- and vertex-capacitated graphs) where cut-trees exist. This model contains both undirected edge cuts and vertex separations where one may construct cut-trees.

2. Directed edge cuts. First we give some notation and the definitions of crossing edge cuts and laminar set systems. The latter is necessary only as background, since the counterexamples can be verified without knowledge of these simple facts.

2.1. Edge cuts. Let $G = (V, E)$ be a graph. For $U \subset V$ let $\delta(U)$ and $\rho(U)$ denote the set of edges leaving (respectively, entering) U. For $V_1 \subset V$, $\delta(V_1)$ is a directed edge cut separating V_1 and $V_2 = V - V_1$. We denote this cut by $(V_1|V_2)$. Two cuts $(V_1|V_2)$ and $(W_1|W_2)$ are *crossing* if the four sets of form $V_i \cap W_j$ are all nonempty.

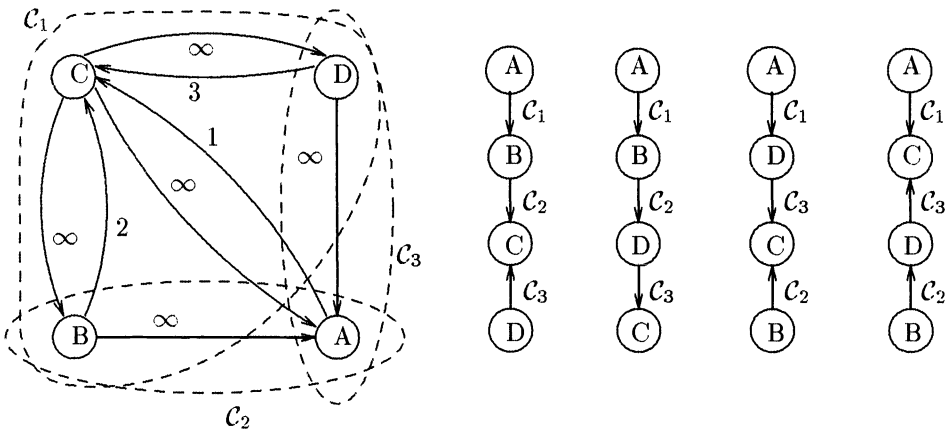


FIG. 1. *Left: directed graph with no cut-equivalent tree. Right: the four flow-equivalent trees of the graph, neither of which is cut equivalent.*

2.2. Set systems. We call a set system *laminar* if for any two nondisjoint members, V_1 and V_2, either $V_1 \subseteq V_2$ or $V_1 \supseteq V_2$. We need two well-known and simple facts. First, for a system of cuts $\{(V_1^{(i)}|V_2^{(i)}) : i \leq n\}$ we can build a laminar system by selecting either $V_1^{(i)}$ or $V_2^{(i)}$ for all $i \leq n$ if and only if it contains no crossing pairs. Second, the laminar systems are those which can be represented as labels of a rooted tree such that the label of a descendant is a subset of that of its ancestor. (This is exactly the case for the Gomory–Hu trees of undirected edge cuts, cf. [9].)

2.3. Counterexamples. Now we give the counterexamples for the erroneous theorems. C.P. Schnorr in [10] claims that it is possible to construct a *cut-tree* in the directed case. We give the counterexample for this theorem in Fig. 1. The graph contains four vertices, hence seven possible cuts. It is easy to find that among them the unique minimum cuts for vertex pairs are $C_1 = (A|BCD)$, $C_2 = (AB|CD)$, and $C_3 = (AD|BC)$ of Fig. 1. Since C_2 and C_3 are crossing, there cannot be a cut-tree for this graph.

Let us discuss the error in Schnorr's paper. His Theorem 3.4 [10, p. 272] is stated as follows. ($F_{u,v}$ is the max-flow value from u *to* v, $\tilde{N}$ is the tree constructed by the algorithm.)

THEOREM 3.4 OF [10]. $\min\{F_{u,v}, F_{v,u}\}$ *equals the minimum capacity of the edges on the path that connects* $\{u\}$ *and* $\{v\}$ *in* $\tilde{N}$. *If among the minimum capacity edges on the undirected path connecting* u *and* v *in* $\tilde{N}$ *some edge* e *is directed from* $\{u\}$ *to* $\{v\}$, *then* $F_{u,v} = \min\{F_{u,v}, F_{v,u}\}$ *and the weak components of* $\tilde{N} - \{e\}$ *yield a minimum* (u, v)*-cut.*

His algorithm does the following. It picks the minimum possible value cut of the graph, on Fig. 1 C_1. Then, as Lemma 3.1 [10, p. 270] states, we can always select minimum value cuts in both components of the previous cut so that they do not cross each other. This is true *inside* $\{B, C, D\}$ since then $C_2 = (B|CD)$ and $C_3 = (D|BC)$ do not cross. However, they do cross in the *entire* graph, which is not recognized in the rest of [10].

Gusfield and Naor have incorrect theorems in [8] because they use the above result of [10]. Their cut-tree construction for the same case as in [10] certainly cannot work. Their other directed cut-tree construction deals with cuts directed *into a fixed vertex*. Adding edges of infinite capacity directed from this fixed vertex to each other vertex makes this equivalent to the original directed problem. Figure 1 with "root" C is a counterexample, because the minimum cuts separating C from the other vertices directed into C are exactly the minimum cuts C_1, C_2, and C_3, where C_2 and C_3 cross each other.

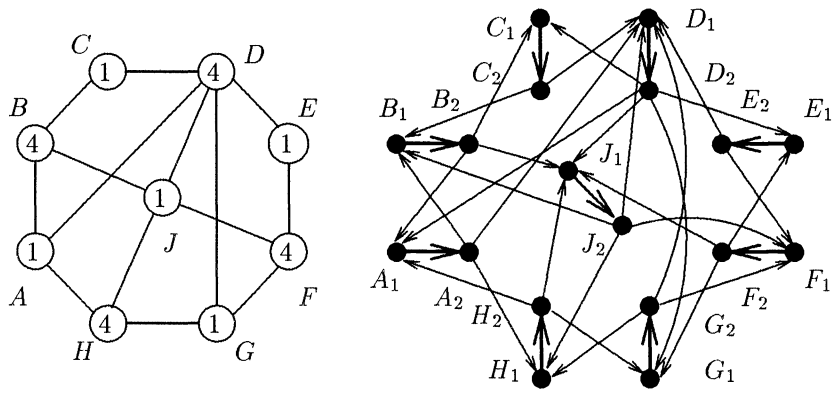


FIG. 2. *Left: vertex-capacitated graph. Right: its transformed digraph.*

3. Undirected vertex cuts. Another application for directed connectivity is to find a compact representation of minimum vertex cuts or cut values: one possible way to deal with vertex cuts is to transform the original graph to a directed graph. In [8] a vertex cut-tree (which is in our terms a cut-tree for separations) algorithm is also presented by transforming the graph to a directed one. Before giving a counterexample to this algorithm, we discuss the method of transforming vertex connectivity in a more general form for mixed cuts as well. Let us start with definitions. Let $G = (V, E)$ be a graph with a positive vertex capacity function c.

DEFINITION OF CUTS. *A* vertex cut *is a minimal $C \subset V(G)$ such that $G - C$ is disconnected. (The number of components can be arbitrarily large.) For a given $T \subset V(G)$, a* mixed cut *(C, D) is a minimal set of vertices $C \subset T$ and edges $D \subset E(G)$ such that erasing them from the graph makes it disconnected. (This is a generalization of vertex cuts, where simply $T = V(G)$ and the edge capacities are infinite.)*

DEFINITION OF SEPARATIONS. *A* separation *denoted by $C = (H_1|C|H_2)$ is a partition of the vertex set $V(G)$ into three (possibly empty) subsets H_1, H_2, and C. C is called the* cutset; *the edges connecting H_1 and H_2 are the* cut edges. *In* vertex *separations there are no edges connecting H_1 and H_2. The capacity of the separation is just $c(C)$.*

We introduce the standard technique (which can be found in [4]) to transform mixed (or vertex) connectivity to directed edge connectivity in another graph $G'(V', E')$ with edge-capacity function e'. Let G' consist of vertices v_1, v_2 for all $v \in V$. We shall denote an edge directed from s to t by (s, t). Then let $(t_1, t_2) \in E'$ with $e'(t_1, t_2) = c(t)$ if $t_1 \neq t_2$. And for all edges (u, v) of G, let (u_2, v_1) and (v_2, u_1) be edges of E' with the same capacity as (u, v) in G. (An example is given in Fig. 2.)

The correspondence of separations and edge cuts is the following. Assume a separation $(H_1|C|H_2)$ is given. Then $H'_j = \{t_j : t \in C\} \cup \{v_1, v_2 : v \in H_j\}$ for $j = 1, 2$ is a partition of V' where the cut directed from H'_1 to H'_2 has the capacity of the separation. Note that we may choose an arbitrary component of the cut for H_1 and let $H_2 = V(G) - C - H_1$. Thus there are always at least two different edge cuts corresponding to the original cut. And conversely, for $s \neq t$ let $(H'_1|H'_2)$ be an s_1–t_2 cut of G'. By defining $H_i = \{v : v_1, v_2 \in H'_i\}$ $(i = 1, 2)$ and $C = V(G) - (H_1 \cup H_2)$, $(H_1|C|H_2)$ is a separation with the same capacity as $(H'_1|H'_2)$.

Note that in the transformed directed graph there are cuts with a single vertex on one side. If we transform them back to the vertex-capacitated graph, a one-vertex "cut" corresponds to these cuts, which "cuts" itself from the rest of the vertices. For vertex pairs connected by an edge, obviously this is the only possibility for a minimum cut. This is why we need the weaker notion of separations: the transformed graph contains information on separations, not

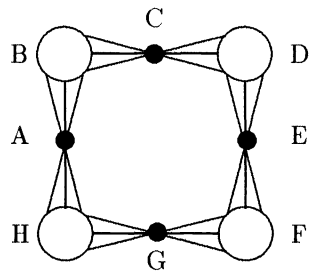


FIG. 3. *Two-vertex-connected graph with two crossing minimum cuts AE and CG.*

cuts. Note that both the (incorrect) result of Gusfield and Naor [8] and the flow-equivalent result of Granot and Hassin [6] deal with separations and not cuts.

Now we discuss a counterexample for the existence of cut-trees for separations in transformed vertex-capacitated graphs. Gusfield and Naor [8] simply use the incorrect result of Schnorr [10] to construct such a tree. Although we have presented a counterexample for Schnorr's result, it is not a transformed graph. Hence we give a special counterexample for this case. Also, it can happen that there is no cut-tree containing a minimum cut for *all* pairs of vertices of the *directed* graph, but if we restrict our attention to a set of edge cuts corresponding to a set of minimum vertex separations for all pairs of vertices, we can build a cut-tree consisting of these cuts. This can happen, since in general there are at least two edge cuts corresponding to vertex cuts and one of them is always redundant. Thus we have an aim which is even stronger than showing that the result of [8] is incorrect: we shall give an example where we cannot select a set of minimum vertex separations for all pairs of nodes and corresponding edge cuts in the transformed graph such that this set of edge cuts bears with a cut-tree representation.

Let us consider the example of Fig. 2. The three cuts of this graph AJC, AJG, and EJG should all be listed to separate vertices B, D, F, and H. Let us see if this is possible on the transformed graph. The two possibilities to select the cut corresponding to the separation ACJ on the transformed graph are $\delta(A_1B_1B_2C_1J_1)$ and $\rho(A_2B_1B_2C_2J_2)$. Let us consider the case when we pick the first one; the analysis of the second case is similar. Then from the two–two possibilities of the cuts corresponding to AJG and EJG, those not crossing the selected one are $\rho(A_2G_2H_1H_2J_2)$ and $\rho(E_2F_1F_2G_2J_2)$, respectively. But these cuts cross each other. So the necessary directed cuts cannot be represented by a tree, contradicting the claim of [8].

Finally, we give an example to show that it is impossible to construct cut-trees for (original) vertex cuts. Let the graph of Fig. 3 have four vertices A, C, E, and G of capacity one; the other vertices are either of large capacity or, if we want unit capacities, they represent large cliques. A cut-tree must contain the two minimum cuts AE and CG to separate C from G (respectively, A from E). If we forget about the four vertices of capacity one, the above two cuts can be considered as edge cuts $(BH|DF)$ and $(BD|HF)$. Since these cuts are crossing each other, there is no cut-tree representing them, i.e., there is no tree which has edges corresponding to both of these two vertex cuts and contains B, D, F, and H as required.

REFERENCES

[1] A. A. BENCZÚR, *Constructing cut-trees in node capacitated and mixed graphs*, manuscript.

[2] C. K. CHENG AND T. C. HU, *Ancestor trees for arbitrary multi-terminal cut functions*. Ann. Oper. Res., 33 (1991), pp. 199–213.

[3] H. FRANK AND I. T. FRISCH, *Communication, Transmission, and Transportation Networks*, Addison-Wesley, Reading, MA, 1971.
[4] L. R. FORD AND D. R. FULKERSON, *Flows in Networks*, Princeton University Press, Princeton, 1962.
[5] R. E. GOMORY AND T. C. HU, *Multi-terminal network flows*. SIAM J. Appl. Math., 9 (1961), pp. 551–560.
[6] F. GRANOT AND R. HASSIN, *Multi-terminal maximum flows in node-capacitated networks*. Discrete Appl. Math., 13 (1986), pp. 143–155.
[7] R. P. GUPTA, *Two theorems on pseudosymmetric graphs*, SIAM J. Appl. Math., 15 (1967), pp. 168–171.
[8] D. GUSFIELD AND D. NAOR, *Efficient algorithms for generalized cut-trees*, Networks, 21 (1991), pp. 505–520.
[9] L. LOVÁSZ AND M. D. PLUMMER, *Matching theory*, North-Holland Math. Stud. 121, Ann. Discrete Math. 29, North-Holland, Amsterdam, 1986.
[10] C. P. SCHNORR, *Bottlenecks and edge connectivity in unsymmetrical networks*, SIAM. J. Comput., 8 (1979), pp. 265–274.

SIAM J. COMPUT.
Vol. 24, No. 3, pp. 511–519, June 1995

008

TREE RECONSTRUCTION FROM PARTIAL ORDERS*

SAMPATH K. KANNAN† AND TANDY J. WARNOW‡

Abstract. The problem of constructing trees given a matrix of interleaf distances is motivated by applications in computational evolutionary biology and linguistics. The general problem is to find an edge-weighted tree which most closely approximates (under some norm) the distance matrix. Although the construction problem is easy when the tree exactly fits the distance matrix, optimization problems under all popular criteria are either known or conjectured to be *NP*-complete. In this paper we consider the related problem where we are given a partial order on the pairwise distances and wish to construct (if possible) an edge-weighted tree realizing the partial order. We are particularly interested in partial orders which arise from *experiments* on triples of species. We will show that the consistency problem is *NP-hard* in general, but that for certain special cases the construction problem can be solved in polynomial time.

Key words. algorithms, graphs, evolutionary trees, evolution

AMS subject classifications. 05C05, 05C85, 06A06, 06A07, 68Q25, 92-08, 92B05, 92D15, 92D20

1. Introduction.

Constructing edge-weighted trees from distance data is a classical problem motivated by applications in molecular biology, computational linguistics, and other areas. Here we are given an n-by-n distance matrix M and asked to find a tree T with leaves $1, 2, \ldots, n$ such that the path distance d_{ij}^T in the tree closely approximates the matrix M. When $d_{ij}^T = M_{ij}$ the matrix is said to be *additive* and efficient algorithms exist for constructing trees from additive distance data (see [8], [5], [1], and others). Various optimization criteria for the problem were proposed and many *NP*-hardness results were published ([2], [3], and others). In fact, in [3] it is shown that for one of the standard optimization criteria (finding a minimum sized tree T such that $d_{ij}^T \geq M_{ij}$, where the size of the tree is the sum of the edge weights in the tree), there is a constant $\epsilon > 0$ such that no polynomial time algorithm can approximate the optimal solution within a ratio of n^ϵ unless *P=NP*. Thus, constructing trees from distance data is a hard problem and the usual heuristic approaches are unlikely to lead to reasonable solutions.

However, for many applications, the actual numeric data is quite unreliable (see [3], [4] for discussions of how interspecies distances are derived in computational molecular biology and why the data is unreliable). One way of handling this unreliability is to assume that distances are given with error bars. This approach was taken by Farach, Kannan, and Warnow in [3]. In this paper, we will take a different approach and assume that we have confidence only in relative information, so that our input will be given in the form of a partial order on the pairwise distances. We are particularly interested in the problem where the partial order is constructed from *experiments* on triples of species, where these experiments are of one of the following two types.

Total order model (TOM). A *TOM* experiment on i, j, k determines the total order of the three pairwise distances $d(i, j), d(j, k), d(i, k)$, with equality or strict inequality indicated.

Partial order model (POM). A *POM* experiment on i, j, k determines the minimum elements of $d(i, j), d(j, k), d(i, k)$.

*Received by the editors July 6, 1993; accepted for publication (in revised form) November 30, 1993.

†Department of Computer and Information Science, University of Pennsylvania, Philadelphia, Pennsylvania. This author's research was supported in part by National Science Foundation grants CCR-8513926 and CCR91-08969.

‡Department of Computer and Information Science, University of Pennsylvania, Philadelphia, Pennsylvania. This author's research was supported by National Science Foundation grant DMS 90-05833 and Department of Energy grants DE-FG03-90ER60999 and DE-AC04-76DP00789. This work began when this author was visiting the Center for Discrete Mathematics and Theoretical Computer Science.

These models are inspired by the model of Kannan, Lawler, and Warnow [6], in which rooted trees are constructed using experiments which determine the *rooted* topology for any three species.

We study the following problems under these models.

Consistency. Given a partial order on a set of distances, does there exist an edge-weighted tree which realizes this partial order?

Construction. Given the ability to perform an experiment, how quickly can we construct an edge-weighted tree realizing the experiments (here we assume the experiments *are* consistent)?

The essential difference between the consistency problem and the construction problem is that we are *not* allowed to perform additional experiments to determine the consistency of a given set of experiments. This makes a substantial difference in complexity, with the consequence that determining consistency is NP-complete for some of the models below where construction can be done efficiently.

We present the following results:

1. The problem of determining whether a set of POM or TOM experiments is consistent with some tree is *NP*-complete, whether the tree is constrained to be unweighted and without degree-two nodes, or can be arbitrary. This result is described in §2.
2. We can construct unweighted binary trees in $O(n^3)$ time from TOM experiments and in $O(n^4)$ time from POM experiments. These results are described in §§3 and 4, respectively.

Constructing unweighted binary trees is motivated by the work of Winkler [9], who considered the related *discrete metric realization* problem, in which one is given an n-by-n distance matrix M and an integer k and the task is to create a graph G with n distinguished vertices and at most k edges such that the shortest path in the graph G between x_i and x_j is exactly equal to M_{ij}. Winkler showed that this problem is *strongly* NP-complete for general graphs and for unweighted graphs without nodes of degree two.

2. Consistency of TOM or POM experiments. In this section we show that determining whether a set of either TOM or POM experiments is consistent with a tree is *NP*-complete for weighted trees as well as for unweighted trees without nodes of degree two. We begin by considering a related problem of constructing trees using unrooted *quartets*, where a quartet is an unrooted tree on four leaves, i, j, k, l. Each quartet q is constrained to contain an edge e so that $q - \{e\}$ describes a partition of the four leaves into two sets of two leaves each. We indicate this by writing $q = (ij, kl)$. Thus, q indicates that the topology on leaves i, j, k, l is as shown in Fig. 1.

The unrooted quartet consistency (UQC) problem is as follows.

Problem: Unrooted quartet consistency.

Input: A set Q of quartets on the species set $S = \{s_1, s_2, \ldots, s_n\}$.

Question: Does there exist a tree T with leaves labeled by the species of S such that if $q = (ij, kl) \in Q$, then there is an edge e in T such that i, j are on one side of e and k, l are on the other side.

The UQC problem was shown *NP-complete* by Steel in [7].

We can now prove that determining consistency of TOM experiments is *NP*-complete.

THEOREM 2.1. *TOM consistency is NP-complete.*

Proof. The reduction is from the UQC problem. Let I be an instance of the UQC problem and let (ab, cd) be one of the topology constraints.

We create two new leaves x and y and write down total order constraints for the triples (x, y, a), (x, y, b), (x, y, c) and (x, y, d) as follows:

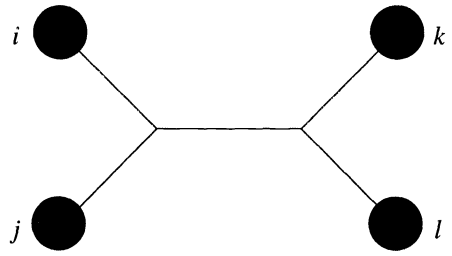


FIG. 1.

$$d(x,a) < d(y,a) < d(x,y),$$
$$d(x,b) < d(y,b) < d(x,y),$$
$$d(y,c) < d(x,c) < d(x,y),$$
$$d(y,d) < d(x,d) < d(x,y).$$

It is not hard to see that any edge-weighted tree satisfying each of the four constraints above also satisfies the topology constraint imposed by the quartet. Thus any tree that is consistent with the constraints of the TOM problem is consistent with the constraints of the UQC problem.

Conversely, suppose there is a tree that is consistent with the constraints of the UQC problem. We augment this tree with the addition of the newly defined leaves such as x and y as follows. For each quartet (ab, cd), let $e = (u, v)$ be an edge in the tree separating ab from cd, with u on the a, b side, and v on the c, d side. In the TOM problem we introduced dummy species x and y such that a, b are closer to x than to y, and c, d are closer to y than to x. Attach these leaves x and y to u and v, respectively and set $w(x,u) = w(y,v) = n^2$. All edges in the original tree are left at unit weight. It is then clear that the augmented tree satisfies all the constraints of the derived instance. Hence we have a valid reduction showing the NP-completeness of the consistency problem for TOM experiments. □

It is easy to see that this proof implies that the consistency of POM experiments is also an *NP*-complete problem, and for both cases the weights on the edges are integers between 1 and n^2. What is not quite as obvious is that consistency of TOM and POM experiments is still *NP*-complete when we restrict ourselves to trees of unit weight and without degree-two nodes. We now prove this for TOM experiments, since the proof for POM experiments follows along the same lines.

THEOREM 2.2. *Determining consistency of TOM experiments with unit weight trees is* NP*-complete.*

Proof. That the problem is in *NP* is trivial; thus we need only show that it is *NP*-hard. By the above proof, determining whether a set of TOM experiments is consistent with a tree in which every edge has integer weight bounded by n^2 is *NP*-complete. We will show that this problem (bounded integer weight TOM experiment consistency) reduces to unit weight TOM experiment consistency.

Let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of leaves and $\mathcal{E}$ a set of k TOM experiments. Let $S' = S \cup \{x_1, x_2, \ldots, x_{2n^3}\}$ be another set of leaves. We will show that $S, \mathcal{E}$ is consistent with a weighted tree T if and only if $S', \mathcal{E}$ is consistent with an unweighted tree T' without degree-two nodes. It is clear that we can assume that T contains no nodes of degree two. So suppose T realizes the TOM experiments. By our construction, we can assume that T has only integer weights on the edges and that these integer weights are bounded by n^2. For each edge e in T with weight $w(e) > 1$, we introduce $w(e) - 1$ additional internal nodes to that edge and $w(e) - 1$ new leaves hanging off those new internal nodes. We then set the weight

of the newly created edges to be 1. It is clear that this new tree T' still satisfies the original set of experiments $\mathcal{E}$ and that it contains at most $2n^3$ additional leaves. Setting $S' = \text{leaves}(T')$, we note that $S', \mathcal{E}$ is realized by T'.

For the converse, if the experiments in E can be realized by a unit-weighted tree T on leaf set S and at most $2n^2$ additional leaves, since E does not contain any constraints involving the additional leaves, E is realized by the tree T with the additional leaves stripped off. □

As a consequence, we have the following corollary.

COROLLARY 2.3. *Let $\leq_P$ be a partial order defined on the set of pairwise distances, d_{ij}. Then determining whether $\leq_P$ is compatible with an edge-weighted (or unweighted) tree is NP-complete.*

3. An $O(n^3)$ algorithm for constructing unweighted trees from TOM experiments. We examine the problem of constructing a tree T with unweighted edges (equivalently, $w(e) = 1$ for all edges $e \in E(T)$) and no nodes of degree two. We will assume here the ability to perform an experiment on any triple of species i, j, k, where the output of the experiment on the triple i, j, k will be a linear ordering on the three pairwise distances, with either equality or strict inequality indicated.

3.1. Sibling sets. The general structure of the algorithm is to discover a pair of sibling supernodes and connect them, making a new supernode. A *supernode* represents a subtree induced by an edge-deletion of T. Note that any tree where each internal node has degree three must have at least two pairs of sibling nodes. Once we locate a sibling pair i_1, i_2 we "collapse" the nodes representing i_1, i_2 into a supernode. We therefore need to show how to determine siblinghood of pairs of supernodes.

We will always maintain the following invariants: we will always know the exact tree that any particular supernode represents and each supernode will represent a portion of the overall tree which is connected to the rest of the tree by exactly one edge. We will abuse the notation slightly and use "supernode" to represent not only a collection of the original vertices, but also the tree structure that we have computed on them. The root of a supernode V is defined to be the node $r(V)$ in V at which the rest of the tree is attached to V. For each supernode V we choose a "representative" v which is a leaf of the original tree that is contained in V and is nearest to the root of V. We denote the representative of a supernode V by $\text{rep}(V)$. We define $d(V) = d(\text{rep}(V), r(V))$. Initially, the supernodes are singleton sets and $d(V) = 0$ for all supernodes V. We will also say that V_1 and V_2 are siblings if $d(r(V_1), r(V_2)) = 2$.

We will clarify each step of the algorithm in what follows. It is particularly important to note that a "collapse" of a set of nodes does not mean throwing away any of the individual nodes; in fact, even after we collapse a set of nodes we will need to refer to experiments involving individual nodes in the set that has been collapsed. Also note that the recursive application of the algorithm is not straightforward since after a single collapse we are not allowed to perform experiments on the new leaf that is created.

3.2. Determining sibling pairs. We have three techniques for determining whether V_1 and V_2 are siblings, depending on how large $|d(V_1) - d(V_2)|$ is.

Given two supernodes V_1 and V_2 with representatives i_1 and i_2, and given any leaf $j \notin V_1 \cup V_2$, we can determine whether j is closer to i_1 or i_2 and use this information to determine whether V_1 and V_2 are siblings. We therefore make the following definitions:

LT: $(V_1, V_2) \in LT$ if and only if there exists leaf $j \in T - (V_1 \cup V_2)$ such that $d(j, i_1) < d(j, i_2)$.

GT: $(V_1, V_2) \in GT$ if and only if there exists leaf $j \in T - (V_1 \cup V_2)$ such that $d(i_2, j) < d(j, i_1)$.

EQ ("equals"): $(V_1, V_2) \in EQ$ if and only if there exists leaf $j \in T - (V_1 \cup V_2)$ such that $d(i_1, j) = d(i_2, j)$.

We now describe the details of how we determine whether two supernodes are siblings. We begin with the case where $d(V_1) = d(V_2)$.

Suppose that $d(V_1) = d(V_2)$ and i_1 and i_2 are the representatives of V_1 and V_2, respectively. If V_1 and V_2 are siblings, then for all $j \notin V_1 \cup V_2$, $d(i_1, j) = d(i_2, j)$. Thus, we will find that $(V_1, V_2) \in EQ$ and $(V_1, V_2) \notin LT \cup GT$. On the other hand, if V_1 and V_2 are not siblings, then there is an internal node v closer to the root of V_1 than to V_2, and if j is a leaf in the subtree rooted at v on the path from V_1 to V_2, then $d(i_1, j) < d(i_2, j)$. Thus, $(V_1, V_2) \in LT$. We therefore can characterize the pairs of supernodes V_1, V_2 such that $d(V_1) = d(V_2)$, which *must* be siblings as explained in the following lemma.

LEMMA 3.1. *Let V_1 and V_2 be supernodes with $d(V_1) = d(V_2)$. Then V_1 and V_2 are siblings* if and only if $(V_1, V_2) \notin LT \cup GT$. *This can be determined in $O(n)$ time.*

Equally simple is the case where $d(V_1) = d(V_2) + 1$. For this case we have the following lemma, the proof of which is omitted.

LEMMA 3.2. *Let V_1 and V_2 be supernodes such that $d(V_1) = d(V_2) + 1$. Then V_1 and V_2 are siblings* if and only if $(V_1, V_2) \notin EQ \cup LT$. *This can be determined in $O(n)$ time.*

In the case where $d(V_1) \geq d(V_2) + 2$, once again let i_1 and i_2 be the representatives of V_1 and V_2, respectively. If $(V_1, V_2) \in EQ \cup LT$ we can conclude immediately that V_1 and V_2 are not sibling supernodes. However, the converse does not hold since if the roots of V_1 and V_2 are connected by a path of length three we would still have $(V_1, V_2) \notin EQ \cup LT$. We need a different technique to handle this case.

We assume that V_1 and V_2 are siblings and derive either a contradiction or confirmation of this assumption. Since we know the entire structure of V_1 and V_2 under our assumption above, we know the length of the path P from i_1 to i_2. We have two cases. If P is of even length, let x be the midpoint of P. Because of our assumption, we can identify the above node x which lies within V_1. Since x has degree greater than two we can find a leaf y which lies in a branch of x other than the ones containing i_1 or i_2 (y lies within V_1 as well and we can identify such a y quickly). Then V_1 and V_2 are siblings if and only if $d(i_1, y) = d(i_2, y)$. If P is of odd length, the proof is similar — we identify the two middle nodes w and x in P and check that for leaves y_1 and y_2 hanging off third branches of w and x, respectively, $d(y_1, i_1) < d(y_1, i_2)$ and $d(y_2, i_1) > d(y_2, i_2)$. Again these conditions will hold if and only if V_1 and V_2 are actually siblings. We summarize the above in the following lemma.

LEMMA 3.3. *Let V_1 and V_2 be supernodes with $d(V_1) \geq d(V_2) + 2$. We can determine whether V_1 and V_2 are siblings in $O(n)$ time.*

Remark 1. It is possible using clever data structures for maintaining the structure of supernodes to reduce the time to check siblinghood in this case to sublinear. However, we don't do this, since in the other cases we do need linear time to check siblinghood.

3.3. Implementation and running time. It is clear that the invariants will be maintained after the detection of sibling pairs and the merger of the constituent supernodes into a bigger supernode. When we have only three supernodes left we stop and construct the tree. We then have to check that this tree is in fact consistent with all of the experiments that have been performed.

In order to see that the running time is $O(n^3)$, note that we perform $O(n^2)$ tests of siblinghood initially (between every pair of given species). Since each test of siblinghood takes $O(n)$ time, this initial step costs $O(n^3)$. Now suppose we determine that two supernodes I and J are siblings. We then need to look at the remaining supernodes to see if any of these are also siblings to I (and hence to J). Hence, to determine the sibling set containing I will cost us $O(n)$ sibling tests, for a total cost of $O(n^2)$. This cost is only incurred by a new sibling pair, and since there are at most $O(n)$ sibling pairs, this only occurs $O(n)$ times. Hence, the overall cost is $O(n^3)$.

Note that the algorithm above is optimal since we have to check that the tree is correct with respect to $\Omega(n^3)$ experiments. However, we might consider a related promise problem where the task is to construct a tree under the promise that such a tree exists. In this case, the lower bound does not hold and the problem of whether there is a more efficient algorithm is open.

4. An $O(n^4)$ algorithm for constructing unweighted binary trees from POM experiments. The algorithm we use for constructing unweighted binary trees from POM experiments uses some of the same techniques as in the previous section. Here, an experiment on i, j, k only returns pair(s) which have minimum distance, and does not totally order that set. We indicate this experimental outcome by $O(i, j, k)$, and understand this to be a set of pairs.

As in the previous algorithm, we work with supernodes and determine siblinghood pairs.

4.1. Determining sibling pairs. We again have techniques for determining siblinghood of supernodes V_1 and V_2, which depend on $|d(V_1) - d(V_2)|$. Since we do not obtain a total order on distances, we use different definitions of the sets *EQ, LT,* and *GT,* in which we may combine information from different experiments in order to infer the existence of internal nodes v for which we can determine the relative proximity of v to the leaves i_1 and i_2, where $i_1 \in V_1$ and $i_2 \in V_2$.

LT: $(V_1, V_2) \in LT$ if and only if there exists leaf $j \in T - (V_1 \cup V_2)$ such that $(i_1, j) \in O(i_1, i_2, j)$ but $(i_2, j) \notin O(i_1, i_2, j)$, *or* there exist leaves j, k in $T - (V_1 \cup V_2)$ such that $(i_1, j) \in O(i_1, j, k)$ and $(i_2, j) \notin O(i_2, j, k)$. *Membership in this set indicates the existence of an internal node v so that $d(i_1, v) < d(i_2, v)$.*

GT: $(V_1, V_2) \in GT$ if and only if there exists leaf $j \in T - (V_1 \cup V_2)$ such that $(i_2, j) \in O(i_1, i_2, j)$ but $(i_1, j) \notin O(i_1, i_2, j)$, *or* there exist leaves j, k in $T - (V_1 \cup V_2)$ such that $(i_2, j) \in O(i_2, j, k)$ and $(i_1, j) \notin O(i_1, j, k)$. *Membership in this set indicates the existence of an internal node v so that $d(i_1, v) > d(i_2, v)$.*

EQ ("equals"): $(V_1, V_2) \in EQ$ if and only if there exists leaf $j \in T - (V_1 \cup V_2)$ such that $\{(i_1, j), (i_2, j)\} \subset O(i_1, i_2, j)$. *Membership in this set indicates the existence of an internal node v so that $d(i_1, v) = d(i_2, v)$.*

In order to be able to use membership or nonmembership in these sets EQ, GT, and LT, we need two more lemmas. First we make the following definition.

DEFINITION 1. *Let i be a leaf, v an internal node and c be a variable or a constant whose value is a positive integer. Suppose it is known that $d(i, v) \geq c$. An experiment on i, r, s is said to be* (c, i, v) critical *if from the outcome of the experiment on (i, r, s) and previous experiments, it is possible to deduce whether $d(i, v) = c$. In other words, an experiment i, r, s is* (c, i, v) critical *if its outcome can be predicted from the assumption that $d(i, v) = c$ and if the predicted outcome will equal the true outcome if and only if $d(i, v) = c$.*

We can now prove the following lemma.

LEMMA 4.1. *Let v be an internal node separating T into three subtrees, T_1, T_2, and T_3. Let i be a node in T_1 with $d(i, v) \geq k$ where k is a constant or a variable whose value is a positive integer and $\min_{x \in \mathcal{L}(T_2)} d(x, v) = t \geq k + 2$. Then there exist leaves $r, s \in T_2$ such that i, r, s is a (k, i, v)-critical experiment.*

Proof. Define the distance function $d^*(i, r) = d(v, r) + k$. Trivially, $d(i, r) \geq d^*(i, r)$ for all $r \in T_2$ with equality if and only if $d(i, v) = k$. Let x be the leaf in T_2 of shortest d^*-distance from i, and let $d^*(i, x) = L$. By construction, $L \geq 2k + 2$. In the path from i to x let c be the node which is at d^*-distance $\lfloor \frac{L}{2} \rfloor$ from i. Since $L \geq 2k + 2$, $c \in T_2$. As before, the removal of c from the tree splits the leaves into three sets, $S_1(c)$, $S_2(c)$, and $S_3(c)$. Let $i \in S_1(c)$ and $x \in S_2(c)$, and let y be the node closest to i in $S_3(c)$. The d^*-distance from y to c is at least equal to $\lceil \frac{L}{2} \rceil$ because of the way that x was chosen. If the d^*-distance from y to

c is less than or equal to $\lceil \frac{L+1}{2} \rceil$, then the experiment i, x, y is a (k, i, v)-critical experiment. Suppose on the other hand that the d^*-distance from y to c is greater than $\lceil \frac{L+1}{2} \rceil$. Then let the path from i to y be of length L' and let c' be the node along this path of d^*-distance $\lfloor \frac{L'}{2} \rfloor$ from i. Then c lies strictly between i and c' and we can repeat the argument by considering i and y and the third branch out of c'. Ultimately this procedure has to terminate since the tree is finite. At this point we will have found suitable x and y so that (i, x, y) is a (k, i, v)-critical experiment. □

Note that in general it is not possible to make the above proof constructive in the obvious manner since we do not know the structure of the overall tree. Thus finding suitable x and y could take $O(n^2)$ time in the worst case.

If V_1 and V_2 were siblings, since we already know the structure of V_1 and V_2 we can predict the outcome of every experiment involving leaves only in $V_1 \cup V_2$. It is thus clear that a necessary condition for V_1 and V_2 to be siblings is that the *predicted* outcome $\mathcal{PO}(i, j, k)$ must equal the actual outcome of the experiment, $O(i, j, k)$, for every $\{i, j, k\} \subset V_1 \cup V_2$.

We also need the following lemma.

LEMMA 4.2. *Let V_1 and V_2 be supernodes with representatives i_1 and i_2, and let $v \notin V_1 \cup V_2$ be a node on the path between i_1 and i_2. Let v separate T into the subtrees $S_t(v)$, $t = 1, 2, 3$ with $i_1 \in S_1(v)$, $i_2 \in S_2(v)$, and let j be the leaf in $S_3(v)$ closest to v. Then there exist leaves $k, l \in T - (V_1 \cup V_2)$ such that the experiments on i_1, i_2, k and l indicate that*

$$(V_1, V_2) \in LT \text{ if } d(i_1, v) < d(i_2, v),$$
$$(V_1, V_2) \in GT \text{ if } d(i_1, v) > d(i_2, v).$$

Proof. If $d(i_1, v) < d(i_2, v)$, then $O(i_1, i_2, j) = \{(i_1, i_2\}$ or else $(i_1, j) \in O(i_1, i_2, j)$. If $(i_1, j) \in O(i_1, i_2, j)$ then $(i_2, j) \notin O(i_1, i_2, j)$ so that $(V_1, V_2) \in LT$. On the other hand, if $(i_1, j) \notin O(i_1, i_2, j)$ then we can deduce that $d(j, v) > d(i_2, v) > d(i_1, v)$ so that $d(j, v) \geq d(i_1, v) + 2$. We can therefore apply Lemma 4.1 and deduce the existence of leaves r, s in $T - (V_1 \cup V_2)$ so that the experiment on (i_1, r, s) is $(d(i_1, v), i_1, v)$-critical. It is then easy to see that the experiments on i_1, i_2, r, and s determine that $d(i_2, v) > d(i_1, v)$ so that $(V_1, V_2) \in LT$. A similar analysis shows that $(V_1, V_2) \in GT$ if $d(i_1, v) > d(i_2, v)$. □

Determining siblinghood when $d(V_1) = d(V_2)$. We can now prove the following lemma.

LEMMA 4.3. *Let V_1 and V_2 be supernodes with $d(V_1) = d(V_2)$. Then V_1 and V_2 are siblings if and only if the ordered pair $(V_1, V_2) \notin GT \cup LT$.*

Proof. The necessity is obvious. For the sufficiency, apply Lemma 4.2. □

Determining siblinghood when $|d(V_1) - d(V_2)| = 1$.

LEMMA 4.4. *Let V_1 and V_2 be supernodes with $d(V_1) = d(V_2) + 1$, with representatives i_1 and i_2, respectively, and assume that the path P between the roots of the supernodes has length at least four. Then there exist experiments indicating that $(V_1, V_2) \in LT$.*

Proof. Let v be the node on P adjacent to the root of V_1. Then $d(v, i_1) = d(V_1) + 1 = d(V_2) + 2 < d(i_2, v)$. By Lemma 4.2, there exist leaves $a, b \in T - (V_1 \cup V_2)$ such that the experiments on a, b, i_1, i_2 indicate that $(V_1, V_2) \in LT$. □

However, we still need to be able to determine that V_1 and V_2 are not siblings when the distance between their roots is exactly three. For this case we have a different technique.

As noted before, if V_1 and V_2 are siblings, then $d(v, i_1) = d(v, i_2) + 1$ for all nodes $v \in T - (V_1 \cup V_2)$. In particular, the parity of the distances $d(v, i_1)$ and $d(v, i_2)$ will be different for every node $v \in T - (V_1 \cup V_2)$. Let P be the path from the root r_1 of V_1 to the root r_2 of V_2. Suppose that P has length exactly three. Let v be the node adjacent to r_1 in P and let x be a leaf in the subtree below v. Then $d(x, i_1) = d(x, i_2)$, so that the parities of $d(x, i_1)$ and $d(x, i_2)$ are identical. We will show that we can determine the existence of a leaf x so that $d(x, i_1)$ and $d(x, i_2)$ have the same parity, when V_1 and V_2 have distance exactly three apart.

LEMMA 4.5. *Let V_1 and V_2 be supernodes of distance three apart and with $d(V_1) = d(V_2) + 1$. Then there exists an experiment indicating that $(V_1, V_2) \in EQ$ or there exist experiments indicating that $parity(i_1, x) = parity(i_2, x)$ for a leaf $x \in T - (V_1 \cup V_2)$.*

Proof. Let v be the node closest to r_1 in the path from V_1 to V_2. As indicated before, every leaf x in the subtree Q rooted at v is equidistant to i_1 and i_2. Suppose for some $x \in Q$, $(i_1, x) \in O(i_1, i_2, x)$. Then $\{(i_1, x), (i_2, x)\} \subset O(i_1, i_2, x)$ so that $(V_1, V_2) \in EQ$. Otherwise, for all $x \in Q$, $(i_1, i_2) = O(i_1, i_2, x)$, so that $d(i_1, i_2) < d(i_1, x) = d(i_2, x)$. Let y be the node of longest distance from i_1 within Q. We will show that we can determine the parity of the distance from i_1 to y.

If the distance $d(i_1, y)$ is even, then the node z halfway along the path from i_1 to y is an element of Q. For any leaf t in the subtree Q' rooted at z not containing either i_1 or y, $O(i_1, y, t) = \{(i_1, t), (y, t)\}$. This indicates that $d(i_1, y)$ is even. On the other hand, if the distance $d(i_1, y)$ is odd, then for all t, $\{(i_1, t), (y, t)\} \not\subset O(i_1, y, t)$. Thus, we can determine the parity of $d(i_1, y)$ for y the leaf furthest from i_1 in Q. However, by construction, y is also the leaf of longest distance to i_2 in Q, so that the parity of $d(i_2, y)$ can also be determined. By our construction, $d(i_2, y) = d(i_1, y)$, so that the parity is the same. □

We summarize our findings in the following theorem.

THEOREM 4.6. *Let V_1, V_2 be supernodes with $d(V_1) = d(V_2) + 1$. Then we can determine in $O(n^2)$ time whether V_1 and V_2 are siblings.*

Proof. We first check whether $(V_1, V_2) \in LT \cup EQ$ by examining $O(n^2)$ experiments involving at least one of i_1 and i_2. If $(V_1, V_2) \in LT \cup EQ$ then we know immediately that V_1 and V_2 cannot be siblings. However, if $(V_1, V_2) \notin LT \cup EQ$, it is still possible that they are not siblings but have a path of distance exactly three between them. By the previous lemma, if the distance between V_1 and V_2 were exactly three, we would be able to determine that $(V_1, V_2) \in EQ$ or else that $d(x, i_1)$ and $d(x, i_2)$ have the same parity, for some $x \in T - (V_1 \cup V_2)$, by examining $O(n^2)$ experiments. Should we find such an x we know that V_1 and V_2 are not siblings. Otherwise the only possibility is that they are in fact siblings. □

Determining siblinghood when $|d(V_1) - d(V_2)| \geq 2$. Determining siblinghood in this case is a straightforward application of Lemma 4.1. More precisely, suppose, without loss of generality that $d(V_1) \geq d(V_2) + 2$. Then we know that $d(r(V_1), i_2) \geq d(V_2) + 2$ with equality if and only if V_1 and V_2 are siblings. Thus we can find x and y (within V_1 in this case) which confirm or refute the assumption of siblinghood of V_1 and V_2.

THEOREM 4.7. *Let V_1 and V_2 be supernodes such that $d(V_2) - d(V_1) \geq 2$, and let $i_1 = \text{rep}(V_1)$. Then V_1 and V_2 are siblings if and only if for all leaves $j, k \in V_2$, $\mathcal{PO}(i_1, j, k) = O(i_1, j, k)$. Furthermore, we can determine whether $\mathcal{PO}(i_1, j, k) = O(i_1, j, k)$ for all $j, k \in V_2$ in $O(|V_2|)$ time, only knowing the structures of V_1 and V_2.* The proof follows that for the case where the experiments are TOM experiments.

4.2. Implementation and running time. At the top level the construction algorithm for the POM model is identical to the algorithm for the TOM model. The difference is only in the procedure for testing siblinghood of two supernodes. In the POM model this takes $O(n^2)$ time leading to an overall running time of $O(n^4)$.

5. Conclusions and open problems. The models presented in this paper strictly generalize any distance-based models of tree reconstruction since we can infer order information given actual distances. It may even be possible to incorporate some tolerance in the distance values by considering $d(i, j)$ to be less than $d(k, l)$ only if $d(k, l) - d(i, j) > B$ for some tolerance parameter B.

There are obvious optimization questions related to the construction questions we have considered; for example, for a given set $\mathcal{E}$ of experiments (TOM or POM), what is the maximum cardinality subset of $\mathcal{E}$ which is consistent with a tree? Since consistency of TOM and POM experiments is an *NP*-complete problem, whether the tree is weighted or unweighted, these problems are *NP*-hard.

The major open question is whether there are polynomial time algorithms to reconstruct *weighted* trees in the TOM and POM models.

REFERENCES

[1] J. CULBERSON AND P. RUDNICKI, *A fast algorithm for constructing trees from distance matrices*, Inform. Process. Lett., 30 (1989), pp. 215–220.

[2] W. H. E. DAY, *Computational complexity of inferring phylogenies from dissimilarity matrices*, Bull. Math. Bio., 49 (1989), pp. 461–467.

[3] M. FARACH, S. KANNAN, AND T. WARNOW, *A robust model for finding optimal evolutionary trees*, Algorithmica, to appear.

[4] J. FELSENSTEIN, *Numerical methods for inferring evolutionary trees*, Quart. Rev. Bio., 57 (1982), pp. 379–404.

[5] J. HEIN, *An optimal algorithm to reconstruct trees from additive distance matrices*, Bull. Math. Biol., 51 (1989), pp. 597–603.

[6] S. KANNAN, E. LAWLER, AND T. WARNOW, *Determining the evolutionary tree*, J. Algorithms, to appear.

[7] M. A. STEEL, *The complexity of reconstructing trees from qualitative characters and subtrees*, J. Classification, 9 (1992), pp. 91–116.

[8] M. S. WATERMAN, T. F. SMITH, M. SINGH, AND W. A. BEYER, *Additive evolutionary trees*, J. Theor. Biol., 64 (1977), pp. 199–213.

[9] P. WINKLER, *The complexity of metric realization*, SIAM J. Discrete Math., 1 (1988), pp. 552–559.

SIAM J. COMPUT.
Vol. 24, No. 3, pp. 520–562, June 1995

009

A GENERALIZATION OF THE SUFFIX TREE TO SQUARE MATRICES, WITH APPLICATIONS*

RAFFAELE GIANCARLO†

Abstract. We describe a new data structure, the *Lsuffix tree*, which generalizes McCreight's suffix tree for a string [*J. Assoc. Comput. Mach.*, 23 (1976), pp. 262–272] to a square matrix. All matrices have entries from a totally ordered alphabet Σ. Based on the Lsuffix tree, we give efficient algorithms for the static versions of the following dual problems that arise in low-level image processing and visual databases.

Two-dimensional pattern retrieval. We have a library of texts $S = \{TEXT^1, \ldots, TEXT^r\}$, where $TEXT^i$ is an $n_i \times n_i$ matrix, $1 \leq i \leq r$. We may preprocess the library. Then, given an $m \times m$, $m \leq n_i$, $1 \leq i \leq r$, pattern matrix PAT, we want to find all occurrences of PAT in $TEXT$, for all $TEXT \in S$. Let $t(S) = \Sigma_{i=1}^{r} n_i^2$ be the size of the library. The preprocessing step builds the Lsuffix tree for the matrices in S and then transforms it into an index (a trie defined over Σ). It takes $O(t(S)(\log|\Sigma| + \log t(S)))$ time and $O(t(S))$ space. The index can be queried directly in $O(m^2 \log|\Sigma| + totocc)$ time, where $totocc$ is the total number of occurrences of PAT in $TEXT$, for all $TEXT \in S$.

Two-dimensional dictionary matching. We have a dictionary of patterns $DC = \{PAT_1, \ldots, PAT_s\}$, where PAT_i is of dimension $m_i \times m_i$, $1 \leq i \leq s$. We may preprocess the dictionary. Then, given an $n \times n$ text matrix $TEXT$, we want to search for all occurrences of patterns in the dictionary in the text. Let $t(DC) = \Sigma_{i=1}^{s} m_i^2$ be the size of the dictionary and let $\bar{t}(DC)$ be the sum of the m_i's. The preprocessing consists of building the Lsuffix tree for the matrices in DC. It takes $O(t(DC) \log|\Sigma| + \bar{t}(DC) \log \bar{t}(DC)))$ time and $O(t(DC))$ space. The search step takes $O(n^2(\log|\Sigma| + \log \bar{t}(DC)) + totocc)$ time, where $totocc$ is the total number of occurrences of patterns in the text.

Both problems have a dynamic version in which the library and the dictionary, respectively, can be updated by insertion or deletion of square matrices in them. In a companion paper we will provide algorithms for the dynamic version.

Key words. image processing, two-dimensional information retrieval, dictionary matching, data structures, pattern matching

AMS subject classifications. 68Q20, 68Q25, 68U15

1. Introduction.

String matching is one of the most widely studied areas in computer science, since it is interesting from a combinatorial point of view and has applications as well. It consists of finding all occurrences of a length m string pat in a length n string $text$, $m \leq n$, where both strings are defined over an alphabet Σ. Aho gives an excellent survey of pattern matching algorithms [1]. Such algorithms are in roughly two complementary classes, with each class satisfying complementary performance criteria and requirements arising in applications.

Dictionary matching algorithms. These algorithms preprocess a dictionary $D = \{pat_1, \ldots, pat_r\}$ of string patterns, each of length m_i; then one can look for the occurrences of patterns of the dictionary into the text. They are appropriate in applications where the dictionary does not change much over time and one can afford slow search times. Aho and Corasick [2] designed the most general algorithms for this class: the dictionary D is preprocessed in $O((\Sigma_{i=1}^{r} m_i) \log|\Sigma|)$ time and then one can search for all the occurrences of patterns of D into a text in $O(n \log|\Sigma| + totocc)$ time, where $totocc$ is the number of such occurrences and n is the length of the text.

Pattern retrieval algorithms. These algorithms preprocess the text to build an index data structure that represents all substrings of the text. The index supports a wide variety of queries. The most basic one is $occurrence(pat)$: report all occurrences of string pat in the text. One can also define more sophisticated queries that ask for statistical information about the structure

*Received by the editors May 28, 1992; accepted for publication (in revised form) December 6, 1993.

†Dipartimento di Mathematica et Applicazioni, Universita di Palermo, Via Archirati 34, Palermo, Italy (raffaele@altair.math.unipa.it). This research was performed while the author was with AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

of the text, e.g., find the longest repeated substring of the text. Such indices are appropriate in applications where we have time to preprocess the text, but we need fast response times to our queries. The best known representative of this class is the suffix tree by McCreight [22] (see also [26] and [10]). It can be built in $O(n \log |\Sigma|)$ time and in $O(n)$ space. The query $occurrence(pat)$ takes $O(m \log |\Sigma| + totocc)$ time. Moreover, it also supports fast answers to statistical queries about the text.

In recent years, there has been growing interest in pattern matching in higher dimensions, due to its relevance to low-level image processing [23] and to the advent of visual databases in multimedia systems [19]. We need the following notation. All matrices have entries defined from a totally ordered alphabet Σ. An $n \times m$ matrix A has height n and width m.

1.1. Two-dimensional pattern matching: State of the art.

Dictionary matching algorithms. The model is similar to that used for strings except that all patterns in the dictionary and the text are matrices. We discuss first the case in which the dictionary is composed of one pattern PAT of dimension $m_1 \times h_1$. Let the text be an $n \times w$ matrix. The first such algorithms, independently discovered by Baker [8] and Bird [9], need $O((m_1 \times h_1 + n \times w) \log |\Sigma|)$ time. Amir, Benson, and Farach [3] made the first substantial step toward alphabet independence for this problem by devising an $O((m_1 \times h_1) \log |\Sigma| + n \times w)$ time algorithm. Recently, Galil and Park [12] closed the gap by improving the time bound to $O(m_1 \times h_1 + n \times w)$. All of these algorithms have the drawback that they can preprocess only one pattern at a time or a dictionary of patterns that can have different widths but the same height. Algorithms for the special case in which the patterns in the dictionary are squares, each of arbitrary height=width, are given by Amir et al. [4], [6] and in this paper (see also [15]). The results have been obtained independently and with different approaches (see §1.2 for technical discussion). Finally, Idury and Schaffer [18] have recently devised data structures and algorithms that preprocess a dictionary of pattern matrices $D = \{PAT_1, \ldots, PAT_r\}$, each of dimension $m_i \times h_i$, $1 \leq i \leq r$; the patterns can be searched for in a text of dimension $n \times w$. The preprocessing step takes $O((\Sigma_{i=1}^{r} m_i \times h_i)\text{polylog}(\Sigma_{i=1}^{r} m_i \times h_i))$ time and the search step takes $O((n \times w)\text{polylog}(\Sigma_{i=1}^{r} m_i \times h_i))$ time.

Pattern retrieval algorithms. The model is similar to that used for strings, but the text is a matrix $TEXT$. Therefore, we want an index that represents all submatrices of $TEXT$ and that at least supports the query $occurrence(PAT)$, i.e., report all occurrences of the pattern matrix PAT in the text matrix. For a long time only straightforward data structures and algorithms were known. Indeed, using a spiral representation of square matrices, Gonnet [16] claims that, for the special case of square matrices, it is possible to build a patricia tree (referred to as PAT-Tree [16]) representing all submatrices of an $n \times n$ $TEXT$ matrix in $O(n^2 \log n)$ *expected time*. However, in the worst case, that data structure can be built in $O(n^4)$ time. Then, using the PAT-Tree, Gonnet [16] claims that one can find all occurrences of a square matrix PAT, of height m, in $O(\log n)$ *expected time* independent of the number of occurrences. Again, in the worst case, the time bound to find all such occurrences is $O(m \log |\Sigma| + occ)$. Here we are interested in the design of data structures that can be efficiently built according to the worst case time analysis. As it will be clear from the results of the next subsection, our data structure guarantees nearly two orders of magnitude speed-up with respect to the solution proposed in [16]. This paper provides the first data structure, the Lsuffix tree, that can be efficiently built and that represents all square submatrices of an $n \times n$ $TEXT$ [15]. Therefore, here we provide a two-dimensional analog of the suffix tree for the special case of square matrices. For completeness, we report that the author has also devised an index data structure that represents all submatrices of a general matrix $TEXT$ [14]. However, it is also shown that the problem of building an index data structure that represents all square submatrices of a square matrix is computationally easier than building an index data structure that represents

all submatrices of a general matrix [14]. Thus the special case we are dealing with here is interesting. Moreover, the algorithmic techniques needed here seem to be interesting in their own right (see §1.2 for technical discussion).

From now on we restrict attention to square matrices.

1.2. Our results. As implied by the above discussion, there was no efficient algorithm known that builds a two-dimensional analog of the suffix tree for square text matrices. The main contribution of this paper is to provide efficient algorithms that build and query such a two-dimensional suffix tree for square matrices, which we call the Lsuffix tree (the name Lsuffix will become clear later). Based on such a data structure, we obtain efficient algorithms for the static version of the following problems that arise in low-level image processing [23] and visual databases [19]. In a companion paper [13], we will provide algorithms for the dynamic version. We show the dependence of the time bound on the alphabet size.

Two-dimensional pattern retrieval. We have a library of texts $S=\{TEXT^1, \ldots, TEXT^r\}$, where $TEXT^i$ is an $n_i \times n_i$ matrix, $1 \leq i \leq r$. We may preprocess the library. Then, given an $m \times m$, $m \leq n_i$, $1 \leq i \leq r$, pattern matrix PAT, we want to find all occurrences of PAT in $TEXT$ (query), for all $TEXT \in S$. Let $t(S) = \Sigma_{i=1}^{r} n_i^2$ be the size of the library.

(i) The preprocessing step builds the Lsuffix tree for the matrices in S and then transforms it into an index (a trie defined over Σ). It takes $O(t(S)(\log|\Sigma| + \log t(S)))$ time and $O(t(S))$ space. Based on the index and in the same time bound, we can precompute three tables which are a space-economical representation of the index taking only a total of $5t(S)$ memory locations.

(ii) The index can be queried directly in $O(m^2 \log|\Sigma| + totocc)$ time, where $totocc$ is the total number of occurrences of PAT in $TEXT$, for all $TEXT \in S$. The query procedure is simple and likely to perform well in practice.

(iii) The tables support a query procedure, with an $O(m^2 + \log t(S) + totocc)$ time bound, which is similar to the one devised by Manber and Myers for suffix arrays [21]. This procedure is also simple and likely to perform well in practice. It beats the one in (ii) when it is costly to store the index or when $|\Sigma|$ is large compared to $t(S)$.

Two-dimensional dictionary matching. We are given a dictionary of patterns $DC = \{PAT_1, \ldots, PAT_s\}$, where PAT_i is of dimension $m_i \times m_i$, $1 \leq i \leq s$. We may preprocess the dictionary. Then, given an $n \times n$ text matrix $TEXT$, we want to search for all occurrences of patterns in the dictionary in the text (search step). Let $t(DC) = \Sigma_{i=1}^{s} m_i^2$ be the size of the dictionary and let $\bar{t}(DC)$ be the sum of the m_i's.

(iv) The preprocessing consists of building the Lsuffix tree for the matrices in DC. It takes $O(t(DC) \log|\Sigma| + \bar{t}(DC) \log \bar{t}(DC)))$ time and $O(t(DC))$ space. Based on it, the search step simulates a finite state automaton in $O(n^2(\log|\Sigma| + \log \bar{t}(DC)) + totocc)$ time, where $totocc$ is the total number of occurrences of patterns in the text. Such a simulation is different from the query procedures in (ii) and (iii).

Amir et al. [4], [6] have independently obtained data structures and algorithms for the two-dimensional dictionary matching problem. In the static version, we have the same time bounds as the ones reported in [6], but the dynamic version of their algorithms is better (see [6], [15]). The algorithms reported in [4] seem to be efficient for the static case only and their time bound is slightly better than the ones obtained here. The algorithmic techniques used in [4], [6] and the data structures they build are different from ours. They build what can be considered the two-dimensional analog of the Aho–Corasick one-dimensional pattern matching machine [2] while we build the two-dimensional analog of the suffix tree [22]. It is an interesting open problem to establish whether their techniques and data structures extend to the efficient construction and query of the Lsuffix tree, so that we can obtain alternative solutions to the two-dimensional pattern retrieval problem.

For the construction of the Lsuffix tree, we introduce Lstrings, a linear representation of square matrices (thus, L stands for linear). Using a different formalism, Amir and Farach [4] introduced such a linearization. We define suffix and prefix relations for Lstrings that are generalizations of the usual ones for strings and that seem to capture the intuitive idea of what a suffix and prefix of a square matrix should be.

The Lsuffix tree is for an Lstring what the suffix tree is for a string. However, because of the new suffix relation, the algorithm that builds the Lsuffix tree is a nonobvious generalization of McCreight's suffix tree algorithm (MC for short) for a string [22]. Indeed, the *rescanning phase*, the key component of MC, breaks down for Lstrings due to the fact that a key property that holds for strings does not hold for Lstrings. Using terminology introduced by Baker [7], we refer to this property as the distinct right context property and we will discuss it in §5. Thus we need new ideas for the design of an efficient algorithm that builds a suffix tree for Lstrings. Another problem is how to label the Lsuffix tree so that it concisely represents all the suffixes of an Lstring (essential to keep the size of the data structures in the pattern retrieval problem bounded by $t(S)$) and how to extract information from those labels (essential to get a fast query algorithm for the pattern retrieval problem). For strings and suffix trees, the problem is easy to solve: a label (i, j) represents the string from position i to j [22]. For Lstrings and their suffixes, it is more complicated since it turns out that it is not convenient to represent Lstrings explicitly as strings in Σ^*.

The algorithmic techniques presented in this paper have already proved useful in other contexts and seem to be able to deal with a general problem that might have other applications as well. Indeed, in a study on the design of tools to detect code duplication in large software systems, Baker [7] has developed a theory for *p-strings*, a generalization of ordinary strings, and she has defined a *p-suffix* tree, a generalization of the suffix tree to *p-strings*. Although far apart in terms of both application areas and definitions, p-strings and Lstrings share the lack of the distinct right context property. Thus the construction of the p-suffix tree uses some of the techniques devised here for the construction of the Lsuffix tree. In general, the techniques presented here are useful when we need to build a suffix-tree-like data structure for objects that are similar to strings except for the fact that the distinct right context property does not hold for them.

We anticipate that the technical presentation of our results will be confined to one matrix. The definitions and algorithms generalize easily to a set of matrices. The remainder of this paper is organized as follows. Section 2 contains some preliminary notation needed in the rest of the paper. In §3 we define Lstrings and point out their relation to matrices. We also define compacted tries for Lstrings and finally the suffix tree for a square matrix A. We briefly discuss the use of such a data structure for pattern matching. In §§5 and 6 we give the algorithm that builds the Lsuffix tree for one Lstring and discuss why the algorithm by McCreight [22] will not work for Lstrings. Then in §7 we generalize the construction to a set of Lstrings and from it we obtain an algorithm to build the suffix tree of a square matrix. In §8 we show how to transform compacted tries for Lstrings into compacted tries for strings, which is needed to obtain fast response times for the pattern retrieval problem. The last three sections deal with applications, concluding remarks, and open problems.

2. Preliminaries. In this section we introduce some notation and the basic data structures needed in the rest of the paper. Given an $n \times n$ matrix A, we denote by $A[i : k, j : l]$ the submatrix of A with corners in $(i, j), (k, j), (i, l), (k, l)$. When $i = k$ or $j = l$, we omit one of the repeated indices. Let Σ be an alphabet and assume that there is a total order $<$ defined on it. Let \$ be a special symbol not part of the alphabet Σ and let us make the convention that it does not match itself, i.e., each instance of \$ is different from the others.

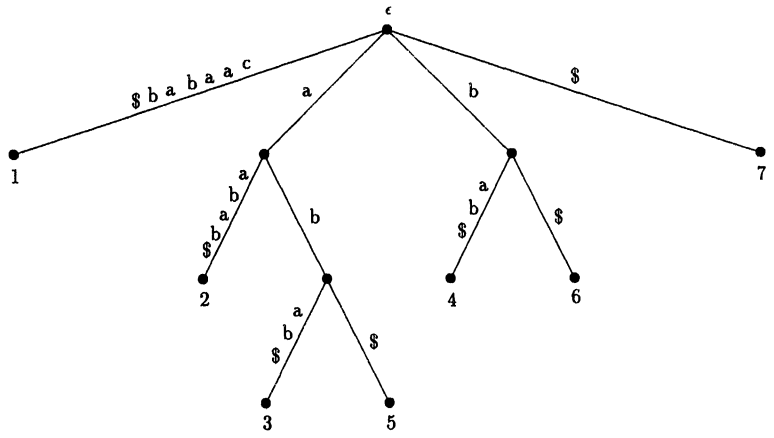


FIG. 1. *Suffix tree for string* caabab$.

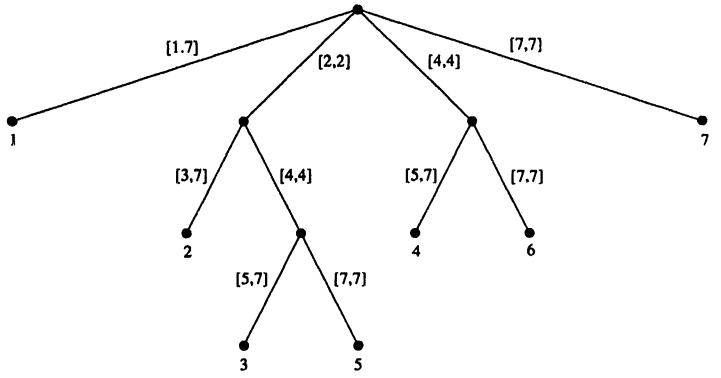


FIG. 2. *Labeled version of suffix tree in Fig.* 1.

Let $x\$$ be a string. The suffix tree T_x for the string $x\$$ [22] is a compacted patricia tree (see [20] for a definition of patricia trees) that satisfies the following conditions. Each edge from parent to offspring is labeled with a substring of $x\$$ and, for each suffix $x[i, |x|]\$$, there is a path from the root to a leaf whose concatenation of labels gives $x[i, |x|]\$$. Because the $\$$ does not match any character of Σ, all suffixes of $x\$$ are distinct. So, there is a one-to-one correspondence between the leaves of T_x and the suffixes of x. We remark that each substring on each edge of the suffix tree can be represented in constant space (see Figs. 1 and 2).

The suffix tree was proposed by McCreight [22] as a space-efficient alternative to Weiner's position tree [26]. It can be built in $O(|x| \log |\Sigma|)$ time and it has $O(|x|)$ nodes. The algorithm by McCreight can also be extended to build the suffix tree for a set of strings $x_1, \ldots, x_k$ by building the suffix tree for $X = x_1\$x_2\$ \ldots \$x_k\$$, where now $\$$ is a separator. The time complexity is $O(|X| \log |\Sigma|)$.

Let F be a forest of rooted directed trees. We are interested in performing the following operations on nodes and edges of F:

- $newnode(v)$: create a new node v, which is also root of a new tree.
- $link(w, v)$: Combine the trees containing v and w by adding the edge (w, v). This operation assumes that v and w are in different trees and that v is a tree root.
- $cut(v)$: Divide the tree containing v into two trees by deleting the edge $(parent(v), v)$. This operation assumes that v is not a tree root.

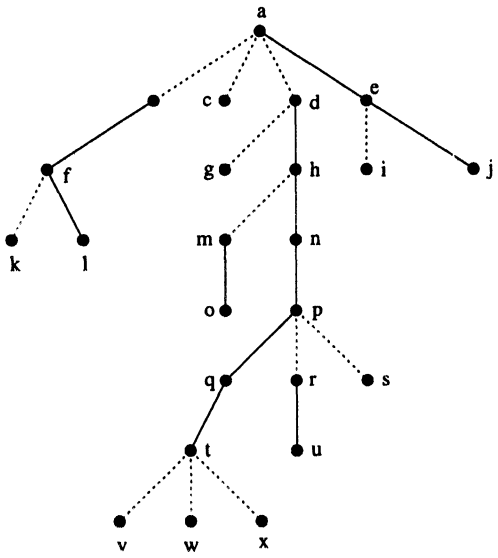


FIG. 3. *A tree partitioned into solid paths. Path t, q, p, n, h, d has head t and tail d.*

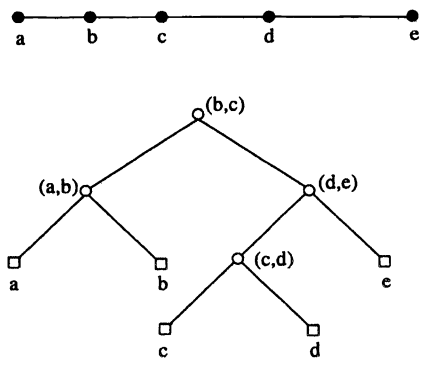


FIG. 4. *A path and a binary tree representing it. Path with head a and tail e. Leaves are labeled with corresponding vertices, internal nodes with corresponding edges.*

Sleator and Tarjan [25] designed a data structure that supports the three operations just defined, among others. Each operation can be implemented to take $O(\log g)$ time in the worst case, where g is the number of nodes in the tree involved in the operation. A tree $\hat{T}$ in the forest is represented as a collection of disjoint solid paths connected by dashed edges (see Fig. 3). Each solid path has a tail and a head, the node further away and closest to the root of $\hat{T}$, respectively (see Fig. 3). What is important for our purposes is that each solid path is represented as a binary search tree such that its leaves, from left to right, give the nodes on the solid path from tail to head. Moreover, the internal nodes represent the edges on the path in such a way that a symmetric order traversal gives the edges on the path from tail to head and, therefore, the nodes in decreasing order of distance from the root of $\hat{T}$ (see Fig. 4). Sleator and Tarjan [25] define the operation $expose(v)$ that returns the path from v to the root of its tree $\hat{T}$ in the forest represented as a solid path. That operation takes $O(\log g)$ time, where g is the number of nodes of $\hat{T}$. For further details, the reader is referred to [25].

We also need the data structures and algorithms of Harel and Tarjan [17] (see also [24]) to compute the lowest common ancestor (LCA for short) of two nodes in a static tree. They

preprocess the tree in time linear in its size. Then, each LCA query can be answered in constant time.

3. From the suffix tree to the Lsuffix tree. In this section we give the definition of Lsuffix tree. We proceed as follows. In §3.1 we discuss at an informal level what such a data structure should represent and how it should represent it. We provide the definition of Lstring and discuss the correspondence between Lstrings and matrices. In §3.2 we formalize the requirements of §3.1. There we first introduce the notion of compacted trie for Lstrings, then give a formal definition of Lsuffix tree for one Lstring. Then we generalize it to a set of Lstrings so that it will represent all square submatrices of a given matrix A and briefly discuss its use for pattern matching.

3.1. Requirements. Let us start with an obvious observation about T_x, the suffix tree of a string $x\$$. As a trie, it represents the set of all suffixes of x in such a way that suffixes with common prefixes share a path in T_x. Since each substring of x is prefix of some suffix of x, for each substring of x there is a path in T_x that corresponds to that substring. That is the reason why T_x is useful for string matching purposes.

Informally, given a square matrix $A[1 : n, 1 : n]$, we want a tree data structure such that for each square submatrix of A there is a path in our tree that "corresponds" to that submatrix, so that we can use it for pattern matching purposes (patterns are square matrices). The tree shape of our data structure seems to impose the constraint that square submatrices of A that have common "prefixes" (whatever the definition of "prefix" is) must share the same path on the tree. Informally, we call it the *common prefix constraint*. If we want a data structure analog to T_x, each submatrix of A must be a "prefix" of some "suffix" of A (whatever the definition of "suffix" is), so that we can build the "correspondence" between all submatrices of A and the paths of our tree by considering only the "suffixes" of A. Informally, we call such correspondence the *completeness* constraint. Our data structure must satisfy both constraints simultaneously.

Keeping those two constraints in mind, let us consider the following definitions of prefix and suffix of a square matrix. For $1 \leq j \leq n$, $A[j : n, j : n]$ is the *jth suffix* of A and $A[1 : j, 1 : j]$ is the *jth prefix* of A (see Fig. 5). Note that any square submatrix of A whose upper left corner lies on the main diagonal of A can be described as a prefix of a suffix of A (see Fig. 5). Let us number each diagonal (not necessarily the main one) of A by d if its elements are $A[i, j]$ with $i - j = d$, $0 \leq |d| \leq n - 1$. Let A_d be the square submatrix of A whose main diagonal is the dth diagonal of A. Since every element of A is on a diagonal of A, it is easy to see that each square submatrix of A is described as the prefix of a suffix of A_d for some d, $0 \leq |d| \leq n - 1$ (see Fig. 5). Thus the completeness constraint can be satisfied provided that our data structure represents all suffixes of A_d, $0 \leq |d| \leq n - 1$.

To satisfy the common prefix constraint, we adopt a linear representation of a matrix A, which we call Lstring. The same representation, with a different formalism, has been introduced by Amir and Farach [4]. We discuss it informally first. Given $A[1 : n, 1 : n]$ and with reference to Fig. 6, notice that we can divide it into n "L-shaped characters", the ith being composed of row $A[i, 1 : i - 1]$ and column $A[1 : i, i]$. Let us "write down" those L-shaped characters in one dimension and in the order given by their top-down appearence in A (see Fig. 6). We get a representation of A in terms of a string of L-shaped characters which we call Lstring. We also need the notion of a chunk, which is the analog of the notion of substring for strings. Informally, we obtain a chunk if we write down the L-shaped characters of A in one dimension, in the order given by their top-down appearence in A and starting at row k and ending at row j (see Fig. 6). As it should be clear from the figure, Lstrings are intended to represent matrices while chunks are intended to represent pieces of matrices centered along the main diagonal.

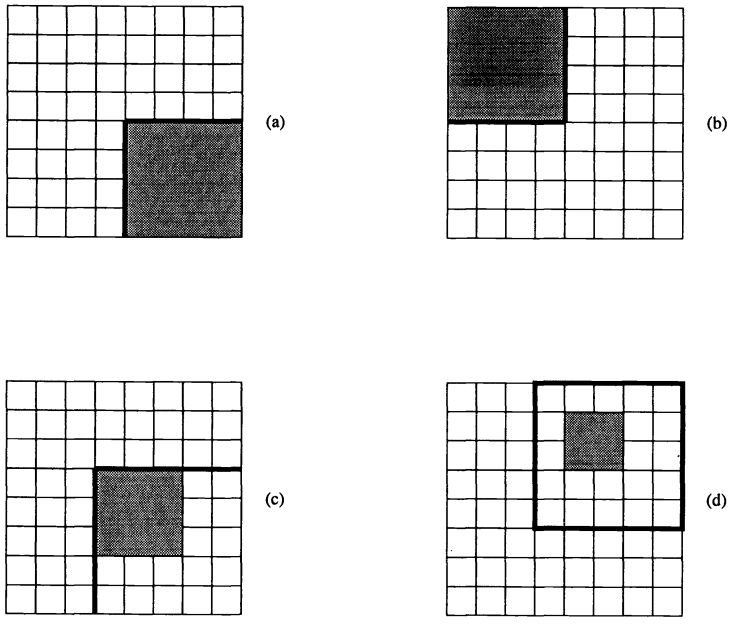


FIG. 5. (a) *The shaded region is the fifth suffix of A.* (b) *The shaded region is the fourth prefix of A.* (c) *The shaded submatrix of A, with upper left corner on the main diagonal of A, is the third prefix of the fourth suffix of A (the one with bold boundaries).* (d) A_{-3} *is shown in bold boundaries. The shaded submatrix of A is the second prefix of the second suffix of* A_{-3}.

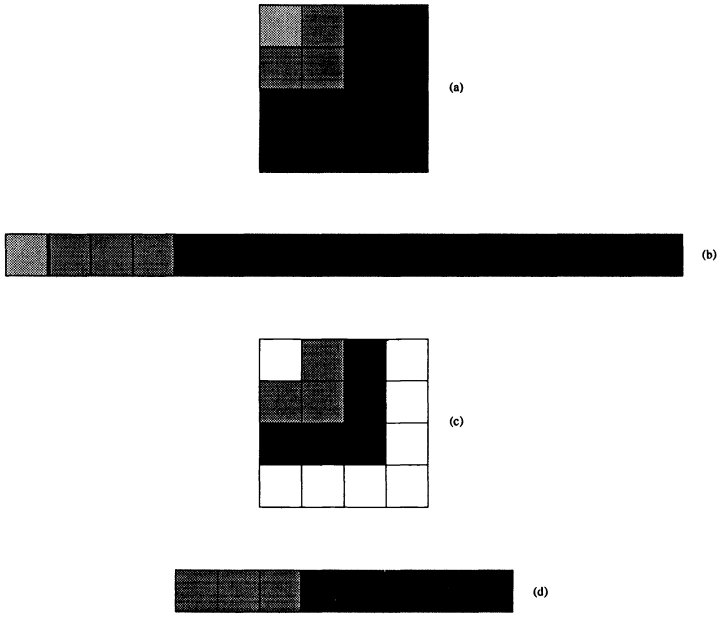


FIG. 6. (a) *A matrix divided into "L-shaped characters." Entries with the same shading are part of the same "character."* (b) *A linear representation of the matrix in* (a) *in terms of "L-shaped characters."* (c) *A chunk composed of two "L-shaped characters," and* (d) *its linear representation.*

More formally, let $L\Sigma = \cup_{i=1}^{\infty} \Sigma^{2i-1}$. We refer to the strings of $L\Sigma$ as *Lcharacters* and we consider each of them as an "atomic" item (composed of "subatomic" parts, which are the

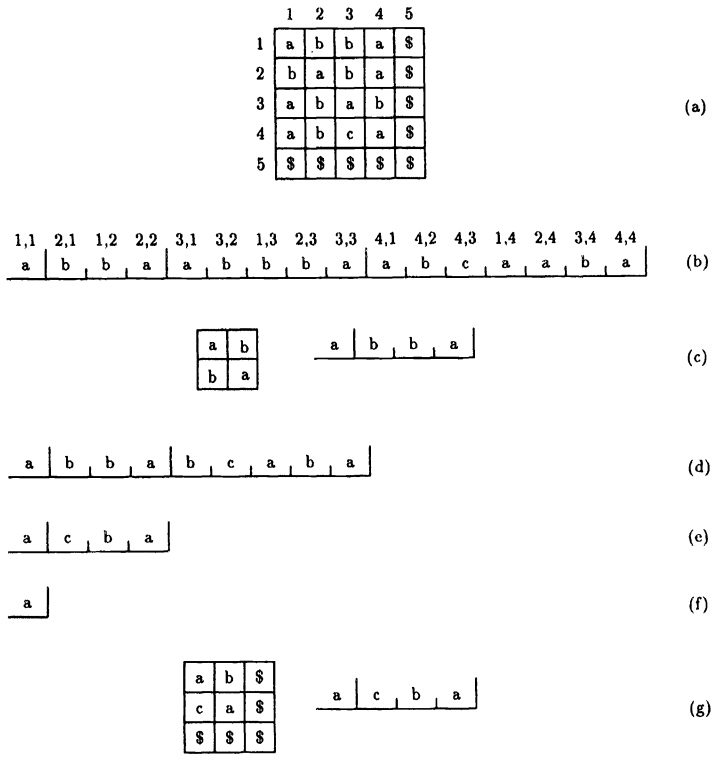


FIG. 7. (b) *Representation of the matrix in* (a), *referred to as A, as an Lstring La. Vertical lines separate Lcharacters. Each letter has on top of it the matrix entry it has been taken from.* \$ *not shown* (c) *A matrix equal to the second prefix of A and its corresponding Lstring equal to the second Lprefix of La.* (d)–(f) *The second, third. and fourth Lsuffix of La, respectively.* \$ *not shown.* (g) *A matrix equal to the third suffix of A and its corresponding Lstring equal to the third Lsuffix of La.* \$ *not shown.*

characters of Σ). We refer to $L\Sigma$ as the *alphabet of Lcharacters*. Two Lcharacters are *equal* if and only if they are equal as strings over Σ. Moreover, given two Lcharacters Lw and Lu of equal length as strings of Σ^*, $Lw \leq Lu$ if and only if Lw as a string is lexicographically smaller than or equal to Lu as a string. Two Lcharacters can be *concatenated* by concatenating the strings corresponding to them; however, we have the following restriction: an Lcharacter in Σ^{2i-1} can *precede* only an Lcharacter in $\Sigma^{2(i+1)-1}$ and *succeed* only one in $\Sigma^{2(i-1)-1}$. Any number of Lcharacters whose concatenation satisfies the above restriction is a *chunk*. An *Lstring* is a chunk such that the first Lcharacter is in Σ. However, the definition of Lstring is most easily understood in terms of their natural correspondence to matrices.

For any matrix $A[1:n, 1:n]$, consider the concatenation of strings $a_1, \ldots, a_n$ such that a_i is a string of length $2i-1$ equal to $A[i, 1:i-1]A[1:i, i]$ (see Fig. 7). Since $a_i \in \Sigma^{2i-1}$, it can be seen as an Lcharacter and, since $a_1 \in \Sigma$, the concatenation of strings $a_1, \ldots, a_n$ is an Lstring La naturally corresponding to A. We denote the ith Lcharacter of La as $La[i]$. The length of La is n, i.e., the number of Lcharacters in the Lstring.

Throughout this paper, when we write down a matrix as an Lstring Lb, the first $i-1$ characters in row i will always correspond to the first $i-1$ characters of $Lb[i]$ and the first i characters of column i will always correspond to the last i characters of $Lb[i]$. Conversely, the first $i-1$ characters of $Lb[i]$ will always correspond to the first $i-1$ characters of row i and the last i characters of $Lb[i]$ will always correspond to the first i characters of column i of a matrix. With such convention, Lb uniquely identifies a square matrix B

and vice versa. A chunk starting at the ith Lcharacter and ending at the jth Lcharacter of Lb is denoted L$b[i, j]$. Through the correspondence between Lstrings and matrices, it is easy to see that L$b[i, j]$ corresponds to a piece of matrix B centered on the main diagonal of B (see Fig. 6).

The jth *Lprefix* of La, denoted L$pref_j(La)$, is the Lstring corresponding to $A[1 : j, 1 : j]$, the jth prefix of A. Given A, L$pref_j(La)$ is easy to obtain since we can take $A[1 : j, 1 : j]$ and write it down as an Lstring. Notice that Lb is an Lprefix of La if and only if the matrix B corresponding to Lb is equal to a prefix of A (see Fig. 7). The jth Lsuffix of La, denoted L$suf_j(La)$, is the Lstring corresponding to $A[j : n, j : n]$, the jth suffix of A. Given A, L$suf_j(La)$ is easy to obtain since we can take $A[j : n, j : n]$ and write it down as an Lstring. Notice that Lb is Lsuffix of La if and only if the matrix B corresponding to Lb is equal to a suffix of A (see Fig. 7).

Let La_d be the Lstring corresponding to A_d. By definition of the prefix and suffix of a matrix, every square submatrix of A is a prefix of some suffix of A_d for some $d, 0 \leq |d| \leq n-1$. Thus, using the correspondence between La_d and A_d and, in general, the correspondence between matrices and Lstrings, we have the following fact.

FACT 1. *Let B be a square matrix and Lb be the Lstring corresponding to it. B is a submatrix of A if and only if Lb is Lprefix of some Lsuffix of La_d, for $0 \leq |d| \leq n-1$.*

Intuitively and drawing an analogy from strings, our data structure needs to be a compacted trie over the alphabet LΣ (whatever such a trie is) that "represents" all Lsuffixes of La_d for all d, $|d| \leq n-1$. That is, for each Lsuffix of La_d, $0 \leq |d| \leq n-1$, there is a path from the root to a leaf "spelling out" that Lsuffix. Using Fact 1 and the fact that matrices with common prefixes are represented by Lstrings with common Lprefixes, that data structure can satisfy both the completeness and the common prefix constraint.

3.2. Definition. We now give a formal definition of our data structure, the Lsuffix tree, and show that it satisfies what we have informally called the completeness and common prefix constraint in the previous section. In §3.2.1 we introduce the notion of trie over the alphabet LΣ. In §3.2.2 we define the Lsuffix tree for one Lstring La and discuss some technical issues related to its representation. Then, in §3.2.3 we generalize it to the set of Lstrings corresponding to A_d, $0 \leq |d| < n$. That data structure will be *the Lsuffix tree* of matrix A.

3.2.1. Tries over the alphabet LΣ. We start with an example. Consider three matrices X, Y, and Z and their corresponding Lstrings Lx, Ly, and Lz (see Fig. 8). We can represent such Lstrings (and therefore the matrices) with a tree by letting Lstrings that have Lprefixes in common share the same path in the tree (see Fig. 8). We can label the edges of such tree with Lcharacters and require that the Lcharacters on two edges, one incoming and the other outgoing the same node, be compatible for concatenation since each of Lx, Ly and Lz must be obtained as the concatenation of the labels on the path from the root to a leaf.

In general, a trie over the alphabet LΣ, representing a set of Lstrings C, is defined in a way analogous to a trie for strings. That is, it is a tree with each edge labeled with an Lcharacter and that satisfies the following:

1. For each Lstring Lx in C there is a leaf v such that the concatenation (according to the rule for Lcharacters) of the labels on the path from the root to v gives Lx.
2. For each node u and w, u is ancestor of w if and only if Lz is the Lprefix of Ly. where Lz and Ly are the Lstrings obtained by concatenating the labels of the edges on the paths from the root to u and w, respectively.
3. All edges outgoing the same node are labeled with a different Lcharacter.

Tries for Lstrings can have nodes of outdegree one, just like tries for strings can have nodes of outdegree one (see Fig. 8). We compact tries for Lstrings by compacting chains of

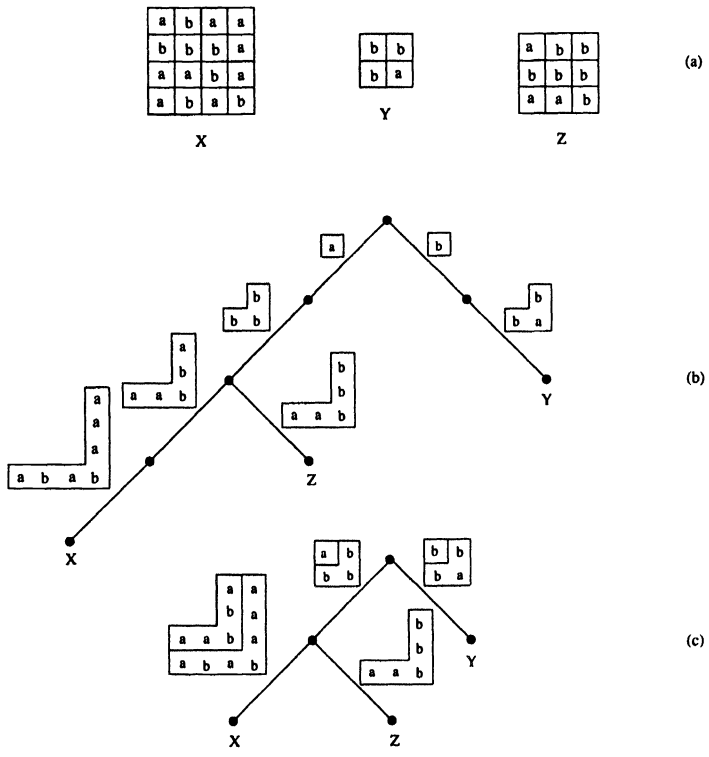


FIG. 8. (b) *The tree over the alphabet* $L\Sigma$ *representing the Lstrings corresponding to matrices in* (a). (c) *The compacted version of the tree in* (b). *Notice chunks on the edges.*

nodes with outdegree one into a single edge. The label on that new edge is the concatenation of the Lcharacters on the edges of the chain, i.e., it is a chunk. So, we can define a compacted trie representing a set of Lstrings as a trie representing the same set of Lstrings in which chains of nodes of outdegree one are compressed into one edge and the edges of the compacted trie are labeled with chunks (see Fig. 8). Notice that there is no loss of information in going from a trie representing a set of Lstrings to a compacted trie representing the same set. Moreover, sibling edges in the compacted trie are labeled with chunks that start with different Lcharacters. We need the following fact.

FACT 2. *Let* C *be a set of Lstrings represented by a compacted trie* LT_C. *Let* Lz *be the Lstring of length* b *obtained by concatenating the chunks on the edges from the root of* LT_C *to some node* v *in* LT_C. *The first Lcharacter on the label of each edge outgoing* v *is a string in* Σ^{2b-1}.

Proof. Notice that in the uncompacted version LT of LT_C, the Lcharacter on the incoming edge of a node $u' \in LT$ at depth i must be in Σ^{2i-1} (see Fig. 8). This follows from the fact that the path from the root of LT to $u' \in LT$ gives an Lstring of length i and the concatenation rule for Lcharacters. Now Fact 2 follows because in LT_C the depth of the nonunary nodes of LT changes but the length of the Lstring they represent is the same. □

We need the following notation. Given a compacted trie LT over the alphabet $L\Sigma$, a node u is the *locus* of an Lstring $L\alpha$ if and only if the concatenation of the labels on the path from the root of LT to u is equal to $L\alpha$. The *extension* of $L\alpha$ is any Lstring of which $L\alpha$ is an Lprefix. The *extended locus* of $L\alpha$ is the locus of the shortest extension of $L\alpha$ whose locus is defined in LT. The *contracted locus* of $L\alpha$ is the locus of the longest Lprefix of $L\alpha$ whose

locus is defined in LT. Using the definitions just given and the definition of a compacted trie over the alphabet $L\Sigma$, the locus, extended locus, and contracted locus of an Lstring are unique nodes in LT. Moreover, when an Lstring has a locus defined in LT then its contracted and extended locuses are the same as its locus. Similar definitions of locus, extended locus, and contracted locus can be defined for a string and a compacted trie over the alphabet Σ.

FACT 3. *Let C be a set of Lstrings and let LT_C be the compacted trie representing the Lstrings in the set. Lstring Ly has an extended locus in LT_C if and only if it is Lprefix of some Lstring in the set C.*

Proof. Assume that Ly has an extended locus u in LT_C. Thus Ly is Lprefix of Lz, the Lstring of which u is a locus. But Lz is the Lprefix of some Lstring in C by definition of trie over the alphabet $L\Sigma$ and the way we compact it. So, Ly is the Lprefix of some Lstring in C.

Assume that Lx is the Lstring in C of which Ly is the Lprefix and let i be the length of Ly, i.e., it is composed of i Lcharacters. Consider the node v on the path from the root to the leaf associated to Lx that is closest to the root and such that the concatenation of the labels on the path p from the root to v gives an Lprefix Lz of Lx of length at least i. Notice that there is no other node $w \neq v$ that is the locus of Lz (otherwise two edges outgoing the same node have labels that start with equal Lcharacters). Therefore v is the extended locus of Ly because Lz is the shortest extension of Ly having a locus defined in LT_C. □

3.2.2. The Lsuffix tree of one Lstring. Intuitively, the Lsuffix tree for one Lstring is like the suffix tree for a string. That is, the Lsuffix tree for an Lstring La is a compacted trie over the alphabet $L\Sigma$ representing the set of all Lsuffixes of La (suffixes of A). Just like the last character of a string is required to be unique for ordinary suffix trees, we require that the last Lcharacter of an Lstring be unique for Lsuffix trees. Given a matrix A, we augment it with a bottom row and rightmost column of \$'s (see Fig. 7a for a matrix so augmented). That bottom row and rightmost column corresponds naturally to an Lcharacter that we denote $\hat{\$}$. The Lstring that corresponds to $A\hat{\$}$ is $La\hat{\$}$. Since \$ does not match anything, no Lsuffix of a given Lstring with $\hat{\$}$ endmarker is an Lprefix of any other Lsuffix of any other Lstring. Formally, the Lsuffix tree LT_a for La is a compacted trie over the alphabet $L\Sigma$, satisfying the following constraints (see Fig. 9 for an example):

1. There is no internal node of outdegree one.
2. Each edge is labeled with a chunk.
3. Chunks assigned to sibling edges start with different Lcharacters, which are of the same length as strings in Σ^*.
4. The concatenation of the chunks labeling the edges on the path from the root to a leaf gives exactly one Lsuffix of $La\hat{\$}$, say $Lsuf_l(La\hat{\$})$ (the Lstring corresponding to $A[l:n, l:n]\hat{\$}$). That leaf is labeled with l.

What is important to note is that there is a one-to-one correspondence between the leaves of LT_a and the Lsuffixes of $La\hat{\$}$ (which are all distinct because $\hat{\$}$ does not match anything).

FACT 4. *Given an $n \times n$ matrix A and its corresponding Lstring La, the Lsuffix tree for La has $O(n)$ nodes.*

Proof. It has exactly $n + 1$ leaves (one per Lsuffix, including $\hat{\$}$) and each internal node has outdegree at least two. □

Notice that each chunk on the edges of LT_a can be a long sequence of Lcharacters and the sum of the lengths of such chunks might exceed $O(n)$, the number of nodes of LT_a. We solve this potential problem by representing each chunk in constant space, independent of its length. The idea is similar to the one used by McCreight in the suffix tree T_x for a string $x\$$ [22], where the substring y on a given edge is represented by a pair (i, j) such that $x[i, j] = y$.

Consider an edge (u, v) in LT_a and let α be the chunk labeling that edge. Let f be any arbitrary leaf in the subtree rooted at v and let l be the label of that leaf. Recall from the

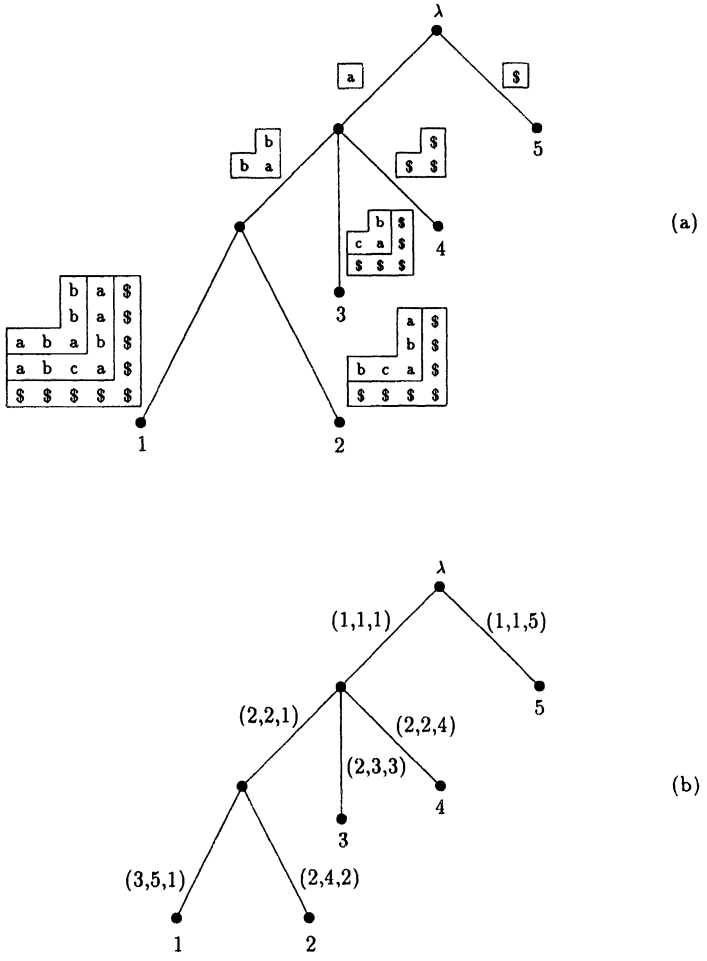


FIG. 9. (a) *The Lsuffix tree corresponding to the Lstring in Fig.* 7b. (b) *The same tree in* (a) *with chunks substituted by triples.*

definition of Lsuffix tree for La that the concatenation of the labels on the path from the root to f must give $Lsuf_l(La\hat{\$})$ (the Lstring corresponding to $A[l : n, l : n]\hat{\$}$). Since (u, v) is on that path, its label α must "appear" somewhere in $Lsuf_l(La\hat{\$})$, i.e., there exist p and q, $p \leq q$, such that the chunk $Lsuf_l(La\hat{\$})[p, q]$ corresponds to α. So, α can be represented in constant space by the triple (p, q, l). In the next paragraph we will discuss how to recover $Lsuf_l(La\hat{\$})[p, q]$ from the triple. We will address the issue on how to compute such triples in §§5 and 6, where we show how to build the Lsuffix tree for one Lstring. For the time being, we assume that each chunk on the edges of LT_a has been substituted by an appropriate triple. See Fig. 9 for an example of an Lsuffix tree for one Lstring in which chunks are represented by triples. From now on, we will make no distinction between chunks and their representation as triples.

One last issue we have to address is how to explictly recover, in constant time, an occurrence in the matrix A of any Lcharacter of the chunk α, when that chunk is represented by the triple (p, q, l). Recall that an Lcharacter is obtained as the concatenation of a subrow and subcolumn of A. What we mean by occurrence of Lcharacter in A is the starting points of a

subrow and subcolumn of A whose concatenation gives the Lcharacter. So, based on the triple, we need to compute the coordinates of two entries of A. Rather than give explicit formulas, we provide a high level description. The triple (p, q, l) corresponds to $\mathrm{L}suf_l(La\hat{\$})[p, q]$. We know that La represents A. Now, $\mathrm{L}suf_l(La\hat{\$})$ =Lc represents a matrix $C = A[l : n, l : n]\hat{\$}$ and so α is $\mathrm{L}suf_l(La\hat{\$})[p, q]$ =Lc$[p, q]$. Thus we can compute in constant time where C starts in A. Because of the correspondence between Lstrings and matrices and the definition of chunk, we know that Lc$[g]$, $p \leq g \leq q$, is equal to $C[g, 1 : g - 1]C[1 : g, g]$. But since we know where C starts in A, we also know where $C[g, 1 : g - 1]$ and $C[1 : g, g]$ start in A and therefore where an occurrence of Lc$[g]$ starts in A. Thus we can access, in constant time, the starting point(s) in A of any Lcharacter in the chunk α based only on the triple (p, q, l). As a side effect of our representation of chunks, we do not need to know explicitly the Lstring La or its Lsuffixes, since we can recover from the matrix A any chunk or Lcharacter in those Lstrings. We obtain the following theorem.

THEOREM 1. *Given an $n \times n$ matrix A, let La be the Lstring corresponding to A. When chunks on the edges of LT_a are represented by triples, the total size of LT_a, i.e., number of nodes, edges, and triples, is $O(n)$. Moreover, given a triple, we can recover in constant time the starting point(s) of an occurrence in A of any of the Lcharacters in the chunk corresponding to the triple.*

Proof. The first part of the theorem comes from Fact 4 and the fact that each edge has only one triple. The second part comes from the discussion preceding the theorem on how to recover chunks from triples. □

3.2.3. The Lsuffix tree for a matrix. Since we need to represent all submatrices of A, we have to define the Lsuffix tree for a set of Lstrings. For notational convenience, we introduce $2n - 1$ special symbols $\$_d$, $0 \leq |d| < n$. Each special symbol is unique and has the same properties of \$. $\hat{\$}_d$, $0 \leq |d| < n$, is analogous to $\hat{\$}$. Let D be the set of Lstrings $La_d\hat{\$}_d$ corresponding to $A_d\hat{\$}_d$, $0 \leq |d| < n$. Formally, the Lsuffix tree LT_D for matrix A and set of Lstrings D is a compacted trie defined on the alphabet $\mathrm{L}\Sigma$ satisfying constraints (1), (2) and (3), of the definition of Lsuffix tree, except that (4) now becomes

4. The concatenation of the chunks labeling the edges on the path from the root to a leaf gives exactly one Lsuffix of the Lstrings in the set, say $\mathrm{L}suf_l(La_i\hat{\$}_i)$. That leaf is labeled with (l, i).

See Fig. 10 for an example. Notice that the length of La_d, i.e., the number of Lcharacters in La_d, is $k_d = n - |d|$ since La_d represents the matrix A_d whose main diagonal is of length $n - |d|$, $0 \leq |d| < n$. Consider the last Lsuffix of $La_d\hat{\$}_d$, i.e., $\mathrm{L}suf_{k_d+1}(La_d\hat{\$}_d)$. It is $\$_d$ because $\$_d$ is the last entry on the main diagonal of $A_d\hat{\$}_d$. By our conventions about $\$_d$ and definition of equality of Lcharacters, the last Lsuffix of $La_d\hat{\$}_d$ is not equal to the last Lsuffix of $La_j\hat{\$}_j$, for any j. So, $2n - 1$ of the leaves of LT_D correspond to $\$_d$, for $1 \leq |d| < n$. Notice also that there is a one-to-one correspondence between the Lsuffixes of $La_d\hat{\$}_d$ and the leaves of LT_D. Let $t(D) = \Sigma_{d=1-n}^{n-1}(k_d + 1)$.

FACT 5. *LT_D has $O(n^2)$ nodes.*

Proof. It has exactly $t(D)$ leaves (one per Lsuffix including $\$_d$'s) and each internal node has outdegree at least two. Moreover $t(D) = O(n^2)$. □

We now prove formally that LT_D satisfies what, in §3.1, we informally called the completeness and common prefix constraint for matrix A.

THEOREM 2. *Let B be a square matrix and let Lb be the corresponding Lstring. B is a submatrix of A if and only if Lb has an extended locus u in LT_D. Moreover, let F be any square submatrix of A having B as a prefix and let Lf be the Lstring corresponding to F. The extended locus of Lf in LT_D is a descendant of u.*

(a)

x	o	o	$
o	x	o	$
x	o	x	$
$	$	$	$

(b)

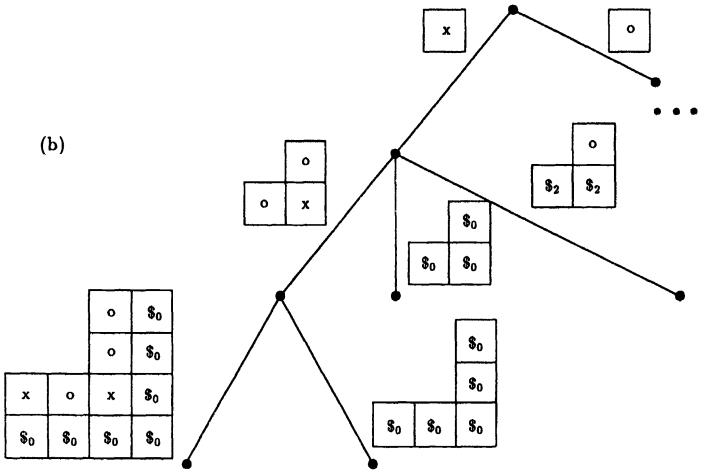


FIG. 10. (b) *The Lsuffix tree of the matrix A in* (a). *Only the suffixes of* A_0 *and* A_2 *are shown in full.*

Proof. By Fact 1, B is a submatrix of A if and only if Lb is an Lprefix of some Lsuffix of La_d, for $0 \leq |d| \leq n-1$. By definition, LT_D is a compacted trie representing the set C of all Lsuffixes of Lstrings in D. So, by Fact 3 (applied to Lb and LT_D) Lb has an extended locus u in LT_D if and only if Lb is an Lprefix of some Lsuffix of La_d, for $0 \leq |d| \leq n-1$. This proves the first part of the theorem.

For the second part, let Lz be the Lstring of which u is the locus. Let us assume that the extended locus v of Lf is not in the subtree of LT_D rooted at u. Since B is the prefix of F, Lb is the Lprefix of Lf. Thus, on the path from the root of LT_D to v, there is a node w that satisfies the definition of extended locus of Lb. $w \neq u$ because v is not a descendant of u. Therefore Lb has two extended locuses, which is a contradiction, since the extended locus of an Lstring is unique. □

Notice that each chunk on the edges of LT_D can be a long sequence of Lcharacters and the sum of the lengths of such chunks might exceed $O(n^2)$, the number of nodes of the Lsuffix tree for the chosen set D and matrix A. We represent each chunk in constant space, independent of its length. The idea is similar to the one used in the case of one Lstring in §3.2.2. Indeed, consider an edge (u, v) in LT_D and let β be the chunk labeling that edge. Let f be any arbitrary leaf in the subtree rooted at v and let (l, i) be the label of that leaf. We can show that β is part of L$suf_l(La_i\hat{\$}_i)$ by reasoning analogous to that used in §3.2.2 to show that α is part of L$suf_l(La\hat{\$})$. Thus the quadruple (p, q, l, i) can represent β, for some $p \leq q$. We will address the issue on how to compute such quadruples in §7, where we show how to build LT_D. For

the time being, we assume that each chunk on the edges of LT_D has been substituted by an appropriate quadruple. From now on, we will make no distinction between chunks and their representation as quadruples (or triples when we are dealing with one Lstring).

Now we show how to explicitly recover, in constant time, the starting point of an occurrence in the matrix A of any Lcharacter of the chunk β, when that chunk is represented by the quadruple (p, q, l, i). The meaning of an occurrence of an Lcharacter in A is the same as the one used for triples. Rather than giving explicit formulas, we provide a high-level description. The quadruple (p, q, l, i) corresponds to $Lsuf_l(La_i\hat{\$}_i)[p, q]$. We know that La_i represents A_i, the square submatrix of A whose main diagonal is the ith of A and we know where A_i occurs in A. The remainder of the proof is analogous to the one used in §3.2.2 to show that we can recover Lcharacters from triples. As a side effect of our representation of chunks, we do not need to know explicitly the Lstrings La_d and their Lsuffixes, $0 \leq |d| < n$, since we can recover from the matrix A any chunk or Lcharacter in them. We obtain the following theorem.

THEOREM 3. *Given an $n \times n$ matrix A, let D be the set of Lstrings corresponding to $A_d\hat{\$}_d$, $0 \leq |d| < n$. When chunks on the edges of LT_D are represented by quadruples, the total size of LT_D, i.e., number of nodes, edges, and quadruples, is $O(n^2)$. Moreover, given a quadruple, we can recover in constant time the starting point(s) of an occurrence in A of any of the Lcharacters in the chunk corresponding to the quadruple.*

Proof. The first part of the theorem comes from Fact 5 and the fact that each edge has only one quadruple. The second part comes from the discussion preceding the theorem on how to recover chunks from quadruples. □

Theorems 2 and 3 can be generalized to hold for a set of matrices $\{A^1, A^2, \ldots, A^s\}$, each of dimension $n_i \times n_i$, $1 \leq i \leq s$. Indeed, D can now be the set of Lstrings corresponding to $A^j_{d_j}\hat{\$}^j_{d_j}$, $1 \leq j \leq s$ and $0 \leq |d_j| < n_j$ ($\$^j_{d_j}$ are special symbols as is \$). Quadruples on each of the edges of LT_D can be trivially substituted by quintuples by adding an entry that keeps track of in which matrix the chunk on that edge occurs (however, we can still use quadruples by suitably renaming the Lstrings in D). The total space is $O(\Sigma_{i=1}^s n_i^2)$.

We now outline how "in principle" LT_D can be used for pattern matching. Assume that we want to find all occurrences of a matrix PAT in A. We write down PAT in terms of the corresponding Lstring $Lpat$. Then, we search for the occurrence of $Lpat$ in LT_D in a way analogous to how we search for the occurrence of a string y in T_x, the suffix tree of a string $x\$$. That is, we find the shortest path starting at the root of LT_D such that the concatenation of the labels on that path gives an Lstring having $Lpat$ as Lprefix. Let v be the last node on that path, i.e., v is the extended locus of $Lpat$ in LT_D. By Theorem 2, PAT occurs in A. Moreover, again by Theorem 2, PAT is the prefix of all suffixes of A_d, $0 \leq |d| < n$, corresponding to the leaves in the subtree of LT_D rooted at v.

To make such approach work in time that is proportional to the size of PAT, we need to reduce the outdegree of the nodes of LT_D. (Given a node $u \in LT_D$, selecting an edge (u, v) whose label starts with a given Lcharacter takes $O(\log n)$ time, since the alphabet $L\Sigma$ is infinite.) In §8 we define the refinement of an Lsuffix tree, the RLsuffix tree, which is a compacted trie defined over Σ and therefore has outdegree at most Σ. Then, in §9 we will show how, based on the search strategy outlined here, we can use the RLsuffix tree for pattern matching.

4. Comparing Lcharacters efficiently. In order to get efficient algorithms for the construction of the Lsuffix tree for a matrix A, we need to be able to compare efficiently two Lcharacters (of equal length as strings in Σ^*). We obtain that by a suitable preprocessing of the matrix A, which we now describe.

Let *rows* (*cols*, respectively) be the string obtained by concatenating the rows (columns, respectively) of A, in row (column, respectively) major order and separated by \$. We build the

suffix trees T_{rows} and T_{cols} for strings $rows$ and $cols$, respectively. That takes $O(n^2 \log |\Sigma|)$ time [22]. Each leaf of T_{rows} (T_{cols}, respectively) is labeled with an entry (i, j) indicating that the path from the root to that leaf spells out $A[i, j : n]\$$ ($A[i : n, j]\$$, respectively). We augment both trees with LCA data structures. That takes $O(n^2)$ time and each LCA query can be answered in constant time [17], [24].

Let us define the operation *compare* as follows. It takes in input the quadruples (p, q, l, i) and (p, q', g, j) and an integer e, $p \leq e \leq \min(q, q')$. It returns the result of the comparison $\mathrm{L}suf_l(a_i\hat{\$}_i)[e] \leq \mathrm{L}suf_g(a_j\hat{\$}_j)[e]$ and the length of the longest prefix common to those two Lcharacters when seen as strings over Σ. We remark that we can define a version of *compare* that takes in input triples rather than quadruples. Its implementation is the same as the one using quadruples. Given the preprocessing of matrix A described above, *compare* can be implemented to take constant time. Let (r_1, c_1) and (r_2, c_2) be the starting points of the subrow and subcolumn of A giving $\mathrm{L}suf_l(a_i\hat{\$}_i)[e]$, i.e., that Lcharacter is equal to $A[r_1, c_1 : c_1 + e - 2]A[r_2 : r_2 + e - 1, c_2]$. Let (r_1', c_1') and (r_2', c_2') be the same points for $\mathrm{L}suf_g(a_j\hat{\$}_j)[e]$, i.e., that Lcharacter is equal to $A[r_1', c_1' : c_1' + e - 2]A[r_2' : r_2' + e - 1, c_2']$. Such starting points can be computed in constant time, as pointed out in §3.2.3. Let f_1 and f_1' (g_1 and g_1', respectively) be the leaves of T_{rows} (T_{cols}, respectively) that have label (r_1, c_1) and (r_1', c_1') ((r_2, c_2) and (r_2', c_2'), respectively), respectively. We use the query $LCA(f_1, f_1')$ (on T_{rows}) to compute the longest prefix common to $A[r_1, c_1 : c_1 + e - 2]$ and $A[r_1', c_1' : c_1' + e - 2]$ and the query $LCA(g_1, g_1')$ (on T_{cols}) to compute the longest prefix common to $A[r_2 : r_2 + e - 1, c_2]$ and $A[r_2' : r_2' + e - 1, c_2']$. That also takes constant time. Given such information, it is easy to compute the output of *compare*. We obtain the following lemma.

LEMMA 1. *We can preprocess the matrix A in $O(n^2 \log |\Sigma|)$ time, so that each compare operation takes constant time.*

Throughout the remainder of this paper, we assume that *compare* takes constant time. We will charge the time complexity of the preprocessing only once to the algorithm of §7.

5. Construction of the Lsuffix tree for one Lstring: High-level description. We give a high-level description of our algorithm for the construction of the Lsuffix tree for $\mathrm{L}a\hat{\$}$, an Lstring of length $n+1$ that corresponds to matrix $A\$$. We do so by drawing an analogy between MC and our algorithm pointing out the differences and presenting the general structure of the latter.

5.1. MC algorithm: High-level description. Let y be a string of length n. MC inserts the suffixes of $y\$$, from longest to shortest, into a tree initially of one node. Let T_i be the suffix tree after the ith insertion (iteration of MC) and let $head_i$ denote the longest prefix of $y[i, n + 1] = y[i, n]\$$ that is also a prefix of $y[j, n + 1]$, for some $j < i$. Since there is a $j < i$ such that $y[j, j + |head_i| - 1] = head_i = y[i, i + |head_i| - 1]$ and $y[j + |head_i|] \neq y[i + |head_i|]$ and T_i is a compacted trie that represents the first i suffixes of $y\$$, $head_i$ has a locus in T_i. Notice that $head_i$ may not have a locus defined in T_{i-1}.

When we insert $y[i, n + 1]$ into T_{i-1} (to transform it into T_i), we can "make it share" for as long as possible a path in T_{i-1} with the suffixes that T_{i-1} represents. Since $head_i$ is the longest prefix that $y[i, n + 1]$ has in common with $y[j, n + 1]$, $1 \leq j < i$, we can proceed as follows.

MC*—High-level description, iteration *i

- Given T_{i-1}, create a locus for $head_i$ in T_{i-1}, if it does not exist, and make a new leaf representing $y[i, n + 1]$ offspring of that locus. (The resulting tree is T_i.)

Unfortunately, at the time of the insertion of $y[i, n + 1]$ into T_{i-1}, $head_i$ is not known and it is not known whether it has already a locus in T_{i-1}. MC cleverly computes where to

possibly create a locus for $head_i$ in T_{i-1} based on $head_{i-1}$ and some additional information computed during previous iterations. We outline the essential parts of such algorithm. For a detailed description, proof of correctness, and time analysis the reader is referred to [22].

Essential to the whole construction is the notion of suffix links with which the suffix tree is augmented and that are incrementally computed by MC. For each internal node p of T_{i-1}, there is a *suffix link* pointing to a node z if and only if p is the locus of a string $a\alpha$, $a \in \Sigma$ and $\alpha \in \Sigma^*$, and $z \in T_{i-1}$ is the locus of α. We now show that, with the exception of the locus of $head_{i-1}$, for each internal node p of T_{i-1} the node where the suffix link of p must point exists in that tree. That will bring to light that the definition of suffix links implicitly uses the distinct right context property for strings defined as follows.

- *Distinct right context property for strings* [7]. In a string x of length n, if the longest prefix that the suffixes $x[i, n]$ and $x[j, n]$ have in common is of length $k+1$, i.e., $x[i, i+k] = x[j, j+k]$ and $x[i, i+k+1] \neq x[j, j+k+1]$, then the longest prefix that the suffixes $x[i+1, n]$ and $x[j+1, n]$ have in common is of length $k \geq 1$, i.e., $x[i+1, i+k] = x[j+1, j+k]$ and $x[i+1, i+k+1] \neq x[j+1, j+k+1]$.

As we will see, such a property does not extend to matrices and Lstrings and therefore we will not be able to define suffix links for the Lsuffix tree. Here is an outline of the proof that the node where the suffix link of p must point exists in T_{i-1}, $p \neq$ locus of $head_{i-1}$. Since $a\alpha$ has a locus p in T_{i-1} and $p \neq$ locus of $head_{i-1}$, it occurs as prefix of at least two of the first $i-2$ suffixes of $y\$$ and in at least two of such occurrences it is followed by distinct letters (recall the definition of T_{i-1}). So $a\alpha$ is the longest prefix that two such suffixes have in common. Applying the distinct right context property to such suffixes (which are among the first $i-2$ of $y\$$), we get that there exist at least two suffixes (now among the first $i-1$ of $y\$$) that have α as the longest prefix. Thus, α must have a locus defined in T_{i-1}, i.e, z exists in T_{i-1}.

Notice that the high-level design of iteration i requires creation of a locus for $head_i$ in T_{i-1} when that locus does not exist. We can proceed in two ways. One consists of finding $head_i$ by starting at the root of T_{i-1} and traversing a path on that tree guided by the characters of $y[i, n+1]$. The other is to use the relationship between $head_{i-1}$, $head_i$ and the nodes of T_{i-1} to skip as much as possible of a prefix of $y[i, n+1]$ and then look for the remainder of $head_i$ by traversing a path of T_{i-1} that starts at a descendant of the root. The second approach will turn out to be more efficient and it is the one followed by MC. To this end, we need to know where to create the locus of $head_i$ in T_{i-1}, when needed.

LEMMA 2 [22]. *Assume that $head_i$ does not have a locus defined in T_{i-1}. The right place in T_{i-1} to create that node is as an offspring of the contracted locus of $head_i$ in that tree.*

When $head_i$ has a locus defined in T_{i-1}, we need to find it because we need to create a leaf offspring of that locus. Otherwise, by Lemma 2, we need to find the contracted locus of $head_i$ in T_{i-1}, create a locus for that string, and then create a leaf as offspring of the new node. Since we do not know whether $head_i$ has a locus in T_{i-1}, we find the contracted locus of that string. By definition of contracted locus and locus of a string, they will be the same node when $head_i$ has a locus defined in T_{i-1}. Again, we want to find such contracted locus by skipping as much as possible of a prefix of $y[i, n+1]$ through the use of the relationship between $head_{i-1}$, $head_i$, and the nodes of T_{i-1}. (As it turns out, all the information needed is already in the tree.) We need the following fact, which can be used to prove the lemma following it. In turn, the lemma tells us the "general neighborhood" in T_{i-1} where the contracted locus of $head_i$ is.

FACT 6 [22]. *Let β be the second suffix of $head_{i-1}$. That is, it is the empty string when $head_{i-1}$ is the empty string and otherwise $head_{i-1} = a\beta$, $a \in \Sigma$ and $\beta \in \Sigma^*$. β is a prefix of $head_i$.*

LEMMA 3 [22]. *The contracted locus of $head_i$ in T_{i-1} is in the subtree rooted at u, where u is the contracted locus of β in T_{i-1} and β is the second suffix of $head_{i-1}$, as defined in Fact 6.*

MC searches for the contracted locus of $head_i$ in T_{i-1} using the strategy suggested by Lemma 3. That is, it locates u and starts the search from u (that amounts to "skipping" a prefix of length $|\beta|$ of $head_i$, during the search for the contracted locus of $head_i$). Then, it modifies T_{i-1} according to the high level design of iteration i. It proceeds as follows.

MC*—Skeleton iteration *i

- If $head_{i-1}$ is the empty string, $u := root(T_{i-1})$; skip *rescanning*.
- *Rescanning phase*: find the locus u in T_{i-1} of β, where $head_{i-1} = a\beta$, $a \in \Sigma$. This is done as follows. Given the parent node v of the locus of $head_{i-1}$ (known to the algorithm at the end of iteration $i-1$), we get to the node w pointed to by the suffix link of v. It can be shown that u is in the subtree of T_{i-1} rooted at w and can be found in amortized constant time by a downward path traversal starting at w. (Thus the suffix link is a shortcut used to find the locus of β quickly. We cannot use the suffix link of the locus of $head_{i-1}$ because it may not be defined in T_{i-1}.)
- *Scanning*: since β is a prefix of $head_i$, find the rest of $head_i$ by traversing the downward path from u in T_{i-1} that "spells out" a prefix of $y[i+|\beta|, n+1]$. (Now, $head_i$ is known and so is its contracted locus in T_{i-1}.) We can create the locus for $head_i$ in T_{i-1} and make a new leaf representing $y[i, n+1]$ offspring of that locus. (Thus T_{i-1} is transformed in T_i.)

5.2. Our algorithm: High-level description. Analogous to MC, our algorithm also inserts the suffixes of $A[1 : n+1, 1 : n+1] = A\hat{\$}$, from longest to shortest, into a tree of initially one node. Recall that such suffixes are $A[j : n+1, j : n+1]$, $1 \le j \le n+1$. More precisely, it inserts the Lsuffixes of $La\hat{\$}$, from longest to shortest, into a tree initially of one node. Let LT_i be the tree at the end of the ith insertion (iteration). That is, LT_i is a compacted trie over the alphabet $L\Sigma$ that represents the suffixes $A[j : n+1, j : n+1]$, $1 \le j \le i$, of $A\hat{\$}$ as Lstrings, i.e., it represents the set of Lsuffixes $Lsuf_j(La\hat{\$})$, $1 \le j \le i$.

Let $Lhead_i$ be the Lstring corresponding to the longest prefix of $A[i : n+1, i : n+1]$ that is also prefix of $A[j : n+1, j : n+1]$, for some $1 \le j < i$, i.e., it is the longest Lprefix of $Lsuf_i(La\hat{\$})$ that is also an Lprefix of $Lsuf_j(La\hat{\$})$, for some $j < i$ (see Fig. 11). In the remainder of this paper, let l_i be the length, i.e., number of Lcharacters, of $Lhead_i$, $1 \le i \le n+1$. Notice that $Lhead_i$ is an Lstring corresponding to a matrix equal to $A[i : i+l_i-1, i : i+l_i-1]$. It is the empty Lstring if and only if $l_i = 0$. Notice also the analogy between $Lhead_i$ and $head_i$.

FACT 7. *$Lhead_i$ has a locus in LT_i.*

Proof. By definition, LT_i is a compacted trie over the alphabet $L\Sigma$ that represents all suffixes $A[k : n+1, k : n+1]$, $1 \le k \le i$, of $A\hat{\$}$ as Lstrings, i.e., all $Lsuf_k(La\hat{\$})$, $1 \le k \le i$. By definition, $Lhead_i$ is the Lstring corresponding to the longest prefix of $A[i : n+1, i : n+1]$ that is also prefix of $A[j : n+1, j : n+1]$, for some $j < i$, i.e., it is the longest Lprefix of $Lsuf_i(La\hat{\$})$ that is also an Lprefix of $Lsuf_j(La\hat{\$})$, for some $j < i$. So, $Lhead_i$ occurs at least twice as an Lprefix of $Lsuf_k(La\hat{\$})$, $1 \le k \le i$, followed by distinct Lcharacters. Therefore, it must have a locus defined in LT_i. □

Note that $Lhead_i$ may not have a locus defined in LT_{i-1}. When we insert the Lstring corresponding to $A[i : n+1, i : n+1]$, i.e., $Lsuf_i(La\hat{\$})$, into LT_{i-1} (to transform it into LT_i), we can "make it share" for as long as possible a path in LT_{i-1} with the Lsuffixes LT_{i-1}. Since $Lhead_i$ is by definition the longest Lprefix that $Lsuf_i(La\hat{\$})$ has in common with the other Lsuffixes represented by LT_{i-1}, we can proceed as follows (notice the analogy with ***MC*—High-level description**):

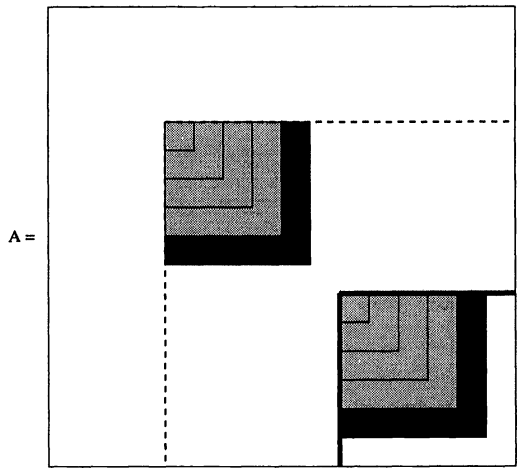


FIG. 11. *$Lsuf_i(La\$)$ corresponds to the suffix of A with bold boundaries. $Lsuf_i(La\$)$ corresponds to the one with dashed boundaries. $Lhead_i$ corresponds to the gray areas. The Lcharacters succeeding the two occurrences of $Lhead_i$ are different.*

Lsuffix tree algorithm—High-level description, iteration *i*

- Given LT_{i-1}, create a locus for $Lhead_i$ in LT_{i-1}, if it does not already exist, and make a new leaf representing $Lsuf_i(La\hat{\$})$, i.e, the suffix $A[i : n+1, i : n+1]$, offspring of that locus. (The resulting tree is LT_i.)

As in ***MC*—High-level description**, at the time of insertion of $Lsuf_i(La\hat{\$})$ into LT_{i-1}, $Lhead_i$ is not known and may not even have a locus defined in that tree. We can try to push the analogy with MC and define suffix links for the nodes of LT_{i-1}. However, the distinct right context property does not extend to matrices and Lstrings. Indeed, it would be as follows. (We state it for matrices but it can be immediately translated into an equivalent form for Lstrings.)

- *Distinct right context property for square matrices.* In a square matrix $A[1 : n, 1 : n]$, if the longest prefix that the suffixes $A[i : n, i : n]$ and $A[j : n, j : n]$ have in common is of length $k+1$, i.e., $A[i : i+k, i : i+k] = A[j : j+k, j : j+k]$ and $A[i : i+k+1, i : i+k+1] \neq A[j : j+k+1, j : j+k+1]$, then the longest prefix that the suffixes $A[i+1 : n, i+1 : n]$ and $A[j+1 : n, j+1 : n]$ have in common is of length $k \geq 1$, i.e., $A[i+1 : i+k, i+1 : i+k] = A[j+1 : j+k, j+1 : j+k]$ and $A[i+1 : i+k+1, i+1 : i+k+1] \neq A[j+1 : j+k+1, j+1 : j+k+1]$.

Figure 12 provides a counterexample. The reader should convince himself or herself that what the lack of such property implies is that if we try to define a suffix link for any node of LT_{i-1} in analogy with the definition by MC, we are not granted the existence in LT_{i-1} of the node where that suffix link should point to. Since suffix links are essential for an efficient rescanning "á la McCreight," we need to come up with a new rescanning technique. For the remainder of this section and for the next one, let $v \in LT_{i-1}$ denote the locus of $Lhead_{i-1}$. Intuitively, our rescanning technique works as follows. We start from v, carefully pick a leaf q in the subtree of LT_{i-1} rooted at v and, from there, "move" to a leaf q' that has the property that the contracted locus of $Lhead_i$ in LT_{i-1} is on the path from the root of LT_{i-1} to q'. Lemmas 4, 5, and 6 prove that such approach works.

Let the leaf $g \in LT_{i-1}$ be the locus of the Lstring corresponding to matrix $A[i-1 : n+1, i-1 : n+1]$, i.e., the locus of $Lsuf_{i-1}(La\hat{\$})$. We define suffix links for all the leaves of LT_{i-1}, except g, as follows.

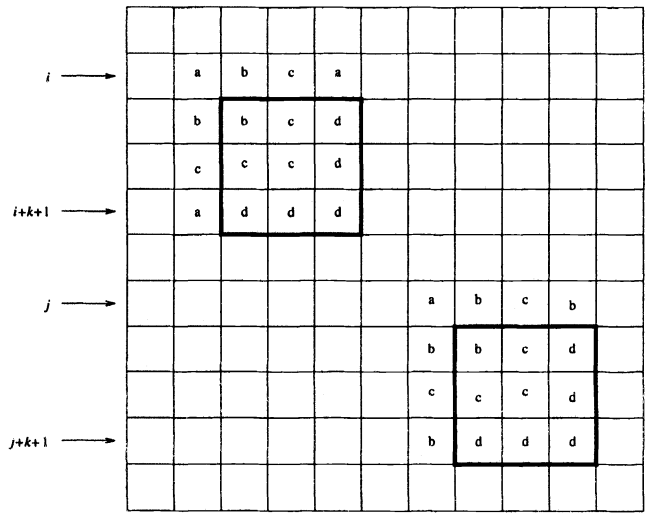


FIG. 12. *A counterexample to the distinct right context property for square matrices. The two submatrices in bold boundaries are equal, whereas they would be different if the property were true.*

DEFINITION 1. *For each leaf* $q \in LT_{i-1}$ *and* $q \neq g$, *locus of the Lstring corresponding to* $A[j : n+1, j : n+1]$, *i.e.,* $Lsuf_j(La\hat{\$})$, $j < i-1$, *there is a suffix link pointing to leaf* w, *where* w *is the locus of the Lstring corresponding to* $A[j+1 : n+1, j+1 : n+1]$, *i.e.,* $Lsuf_{j+1}(La\hat{\$})$. *We denote such suffix link as* $SL(q) = w$.

Note that the high-level design of iteration i requires creation of a locus for $Lhead_i$ in LT_{i-1} when that locus does not exist. As for strings, we can proceed in two ways. One consists of finding $Lhead_i$ by starting at the root of LT_{i-1} and traversing a path on that tree guided by the Lcharacters of $Lsuf_i(La\hat{\$})$. The other is to use the relationship between $Lhead_{i-1}$, $Lhead_i$, and the nodes of LT_{i-1} to skip as much as possible of an Lprefix of $Lsuf_i(La\hat{\$})$ and then look for the remainder of $Lhead_i$ by traversing a path of LT_{i-1} that starts at a descendant of the root. The second approach will turn out to be more efficient and it is the one our algorithm will follow. To this end, we need to know where to create the locus of $Lhead_i$ in LT_{i-1}, when needed.

LEMMA 4. *Assume that* $Lhead_i$ *does not have a locus defined in* LT_{i-1}. *The correct place in* LT_{i-1} *to create that node is as an offspring of the contracted locus of* $Lhead_i$ *in that tree. The extended locus of* $Lhead_i$ *in* LT_{i-1} *exists and it is an offspring of the contracted locus of* $Lhead_i$ *in that tree.*

Proof. Since $Lhead_i$ does not have a locus defined in LT_{i-1}, it cannot be the empty Lstring (otherwise $root(LT_{i-1})$ would be its locus). Thus, the matrix $E = A[i : i+l_i-1, i : i+l_i-1]$ corresponding to it cannot be empty. By definition of $Lhead_i$, the correspondence between Lstrings and matrices and the fact that \$ does not match any character (including itself), there exists a $j < i$ such that E is a proper prefix of $W = A[j : n+1, j : n+1]$. Thus $Lhead_i$ is a proper Lprefix of $Lsuf_j(La\hat{\$})$, $j < i$, the Lstring corresponding to a matrix equal to W. By Fact 3 (applied to LT_{i-1} and $Lhead_i$), $Lhead_i$ has an extended locus $\hat{v}$ in LT_{i-1}. Since $Lhead_i$ does not have a locus defined in LT_{i-1}, the contracted locus v' of $Lhead_i$ in LT_{i-1} must be the parent of $\hat{v}$ (by the definition of contracted and extended locuses of an Lstring). Thus the locus of $Lhead_i$ must be an offspring of v' since the Lstring of which v' is locus is the longest Lprefix of $Lhead_i$ that has a locus defined in LT_{i-1}. We create such locus by "splitting" the edge $(v', \hat{v})$. □

When $L head_i$ has a locus defined in LT_{i-1}, we need to find it because we need to create a leaf offspring of that locus. Otherwise, by Lemma 4, we need to find the contracted locus of $L head_i$ in LT_{i-1}, create a locus for that Lstring, and then create a leaf as an offspring of the new internal node. Since we do not know whether $L head_i$ has a locus in LT_{i-1}, we find the contracted locus of that Lstring. When $L head_i$ has a locus defined in LT_{i-1}, they will be the same node by the definition of contracted locus and locus of an Lstring. Again in analogy with MC, we want to find such a contracted locus by skipping as much as possible of an Lprefix of $L suf_i(La\hat{\$})$ through the use of the relationship between $L head_{i-1}$, $L head_i$, and the nodes of LT_{i-1} (as it turns out, all the information needed is already in LT_{i-1}). In this section and the next one, let $h = \min(0, l_{i-1} - 1)$. Notice that the matrix $A[i : i + h - 1, i : i + h - 1]$ corresponds to $L suf_2(L head_{i-1})$. It is empty if and only if $L suf_2(L head_{i-1})$ is the empty Lstring, i.e., $0 \leq l_{i-1} \leq 1$. In the remainder of this paper, let $L\alpha$ denote $L suf_2(L head_{i-1})$. We need the following lemma, which states some properties of $L\alpha$.

LEMMA 5. *$L\alpha$, the Lstring corresponding to $A[i : i + h - 1, i : i + h - 1]$, is such that* (a) *it is an Lprefix of $L head_i$, the Lstring corresponding to $A[i : i + l_i - 1, i : i + l_i - 1]$;* (b) *it has an extended locus r' in LT_{i-1};* (c) *r' is an ancestor of any leaf $\hat{u} = SL(q)$, where $q \neq g$ is any leaf descendant of v; and* (d) *the contracted locus u of $L\alpha$ in LT_{i-1} is the parent of r', when $L\alpha$ does not have a locus defined in LT_{i-1}.*

Proof. By the definition of $L head_{i-1}$, there is a suffix $A[j : n+1, j : n+1]$, $j < i-1$, of $A\hat{\$}$ such that the longest prefix common to $A[j : n+1, j : n+1]$ and $A[i-1 : n+1, i-1 : n+1]$ is $A[i - 1 : i + l_{i-1} - 2, i - 1 : i + l_{i-1} - 2]$ (it may be empty). Let q be the leaf locus of $L suf_j(La\hat{\$})$. Since $L head_{i-1}$ is an Lprefix of $L suf_j(La\hat{\$})$, q is in the subtree of LT_{i-1} rooted at v. Moreover, it cannot be equal to g because that leaf is locus of $L suf_{i-1}(La\hat{\$})$ and $j < i - 1$. Let $\hat{u} = SL(q)$. By definition of a suffix link, $\hat{u}$ is the locus of $L suf_{j+1}(La\hat{\$})$.

Since $h = \min(0, l_{i-1}-1)$, $A[j+1 : n+1, j+1 : n+1]$ has $A[i : i+h-1, i : i+h-1]$ as a prefix (it may be empty). So $L\alpha$, the Lstring that corresponds to $A[i : i+h-1, i : i+h-1]$, is an Lprefix of $L head_i$ by definition of $L head_i$. That proves (a).

As for (b), (c), and (d), note that $L\alpha$ is the Lprefix of $L suf_{j+1}(La\hat{\$})$ ($A[j+1 : n+1, j+1 : n+1]$ has $A[i : i + h - 1, i : i + h - 1]$ as a prefix). So, by Fact 3 (applied to $L\alpha$ and LT_{i-1}), $L\alpha$ has an extended locus r' in that tree. Moreover, r' must be the locus of an Lstring Lprefix of $L suf_{j+1}(La\hat{\$})$ (otherwise it cannot be the extended locus of $L\alpha$). Therefore, r' is an ancestor of $\hat{u}$. The contracted locus u of $L\alpha$ is either r' or the parent of r' in LT_{i-1}, by definition of the contracted locus, extended locus, and locus of an Lstring. □

Note that the contracted locus of an Lstring always exists in a compacted trie over the alphabet $L\Sigma$ since the root is the locus of the empty Lstring. In the remainder of this section and in the next one, let u denote the contracted locus of $L\alpha$ in LT_{i-1}, Lb denote the string of which u is locus, and r' denote the extended locus of $L\alpha$ in LT_{i-1}. The following lemma tells us the "general neighborhood" in LT_{i-1} where the contracted locus of $L head_i$ is as given in the following lemma.

LEMMA 6. *The contracted locus of $L head_i$, the Lstring corresponding to $A[i : i+l_i-1, i : i + l_i - 1]$, in LT_{i-1} is in the subtree rooted at u.*

Proof. Lb is the Lstring obtained by concatenating the labels of the path from $root(LT_{i-1})$ to u. By definition of a contracted locus, Lb is an Lprefix of $L\alpha$ and therefore an Lprefix of $L head_i$. (By Lemma 5, $L\alpha$ is Lprefix of $L head_i$.) But only the nodes in the subtree rooted at u have Lb as Lprefix (by definition of compacted trie over the alphabet $L\Sigma$). Therefore the contracted locus of $L head_i$ must be in that subtree. □

In analogy with MC, we search for the contracted locus of $L head_i$ in LT_{i-1} using the strategy suggested by Lemma 6. That is, we locate u and start the search from u. We will

show that, in such a way, we can "manage to skip" Lα, an Lprefix of L$head_i$, when searching for the contracted locus of L$head_i$ in LT_{i-1}. Then we modify LT_{i-1} according to the high-level design of iteration i. We also have a rescanning and scanning phase. As with MC, our rescanning phase finds u. However, it is implemented differently than the one in MC because we have suffix links defined at the leaves of LT_{i-1} whereas MC has suffix links defined for the internal nodes of T_{i-1}. In addition, we need to be careful with the management of the labels on the edges of the Lsuffix tree, the comparisons of Lcharacters, and the update of our auxiliary data structures. (We will deal with such issues in §6.)

We anticipate that our procedure will not do rescanning when L$head_{i-1}$ is the empty Lstring because in that case L$suf_2(Lhead_{i-1}) =$ Lα is the empty Lstring and its contracted locus (and locus) is $root(LT_{i-1})$, so it is already known. The skeleton of the ith iteration of **STI**, the procedure that inserts Lsuffixes into the Lsuffix tree, is given below. We assume, and justify later, that it knows the locuses v and g in LT_{i-1} of the Lstrings corresponding to $A[i-1 : i+l_{i-1}-2, i-1 : i+l_{i-1}-2]$ and $A[i-1 : n+1, i-1 : n+1]$, i.e., L$head_{i-1}$ and L$suf_{i-1}(La\hat{\$})$. For the first iteration, those two nodes are the same (the root of LT_0).

Procedure STI—Skeleton iteration i

1. If $v = root(LT_{i-1})$ then $u := root(LT_{i-1})$ and skip *rescanning*.
2. *Rescanning*: find the contracted locus u of Lα, given the locus $v \neq root(LT_{i-1})$ of L$head_{i-1}$ and SL for all leaves in LT_{i-1}, except g.
3. *Scanning*: starting from u and skipping Lα, find the contracted locus of L$head_i$. If needed, create a locus for L$head_i$ and make a new leaf representing the $Lsuf_i(La\hat{\$})$ offspring of that locus. Compute SL for g.

6. Construction of the Lsuffix tree for one Lstring: Detailed description. In this section we show how we proceed for the rescanning and scanning phase of the ith iteration of our algorithm. We do so by assuming the following invariant, which will be maintained by the algorithm and tells us which information we have available at the beginning of iteration i to perform the computation prescribed by the skeleton of **STI**. Before we state the invariant, we recall that the leaf $g \in LT_{i-1}$, the locus of the Lstring corresponding to matrix $A[i-1 : n+1, i-1 : n+1]$, does not have a value of SL defined.

INVARIANT 1. *At the beginning of iteration i,* **STI** *knows the locus v of L$head_{i-1}$ and the leaf g locus of $Lsuf_{i-1}(La\hat{\$})$ in LT_{i-1}. The values of SL are known to* **STI** *for all leaves in LT_{i-1}, except g. For all nodes z of LT_{i-1}, the length $l(z)$ of the Lstring of which z is locus is known to* **STI**. *Each internal node $c \in LT_{i-1}$ has a pointer to a leaf (arbitrarily chosen) in the subtree of LT_{i-1} rooted at c. Moreover, all edges outgoing any internal node of LT_{i-1} are sorted according to the first Lcharacter on the chunk labeling that edges (we keep them in a binary search tree).*

Based on the information granted by Invariant 1 we have to show the following:

1. How to find the contracted locus u of Lα, when $v \neq root(LT_{i-1})$. (This corresponds to rescanning.)
2. How to find, starting from u, the contracted locus of L$head_i$. (This is part of scanning.)
3. How to change LT_{i-1} into LT_i, computing, for the new nodes the length of the Lstrings of which they are locuses and, for the new edges, their labels. How to compute $SL(g)$. (This is part of scanning.)

6.1. Point one. We refer to the operation that finds the contracted locus u of Lα in LT_{i-1} as $findclocus$. It takes in input $v \neq root(LT_{i-1})$, the locus of L$head_{i-1}$ in LT_{i-1}. We first prove its correctness (Lemma 7) and then we discuss its implementation (Lemma 8).

Procedure *findclocus*

- Select any offspring c of v not equal to the leaf g locus of $\mathrm{L}suf_{i-1}(La\hat{\$})$. Let q be the leaf of $\mathrm{L}T_{i-1}$ pointed to by c. On the path from the root of $\mathrm{L}T_{i-1}$ to $SL(q)$, u is the deepest node such that $l(u) \leq h$.

LEMMA 7. *Assume that $v \neq root(LT_{i-1})$ and that Invariant 1 holds at the beginning of iteration i. findclocus correctly computes the contracted locus u of $L\alpha$ in LT_{i-1}.*

Proof. Notice that by the definition of $\mathrm{L}head_{i-1}$ and the fact that g is the locus of $\mathrm{L}suf_{i-1}(La\hat{\$})$, g is an offspring of v, so the node c selected by *findpath* is not an ancestor of g. By the invariant, c points to a leaf q in the subtree of $\mathrm{L}T_{i-1}$ rooted at c. Since c is not an ancestor of g, $q \neq g$. Therefore, $\hat{u} = SL(q)$ is defined by the invariant and can be computed by *findclocus*.

Since $q \neq g$ is any arbitrary leaf in the subtree of $\mathrm{L}T_{i-1}$ rooted at v, we have that the extended locus r' of $\mathrm{L}\alpha$ is an ancestor of $\hat{u}$ (by Lemma 5). By Lemma 5, u is an ancestor of r' and therefore an ancestor of $\hat{u}$. Thus, by the definition of contracted locus, we have that u must be the deepest node on the path from $root(LT_{i-1})$ to $\hat{u}$ such that $l(u) \leq h$. □

In order to efficiently perform *findclocus*, we represent $\mathrm{L}T_{i-1}$ as a dynamic tree [25]. That is, as a set of edge-disjoint solid paths. *Expose* is the dynamic tree operation of interest to us for the implementation of *findclocus*. It has been described in §2. We implement *findclocus* as follows. Let $\hat{u} = SL(q)$. Since Invariant 1 holds, we can get from v to $\hat{u}$ in constant time. We do $expose(\hat{u})$: it returns the path $\hat{p}$ from $\hat{u}$ to the root of $\mathrm{L}T_{i-1}$ in a search tree. As pointed out in §2, a symmetric order traversal of that search tree gives the nodes of $\hat{p}$ in decreasing order of distance from $root(LT_{i-1})$. Note that $l(w) < l(y)$, when node w is closer than y to the root. Moreover, by the invariant, for each node of $\hat{p}$, the length of the Lstrings of which it is locus is known to **STI**. Thus we can use that binary search tree to find, by binary search, the node u further away from $root(LT_{i-1})$ such that $l(u) \leq h$.

Since LT_{i-1} has at most $O(n)$ nodes (at most as many as the final Lsuffix tree for $La\hat{\$}$), *expose* takes $O(\log n)$ time on a dynamic tree of size at most $O(n)$ [25]. Moreover, the binary search is performed on a balanced search tree of at most $O(n)$ nodes, so *findclocus* takes $O(\log n)$ time. For later reference, we summarize the above discussion in the following lemma.

LEMMA 8. *Assume that the invariant holds at the beginning of iteration i. findclocus can be implemented to take $O(\log n)$ time.*

6.2. Point two. In what follows, let B denote the matrix corresponding to the Lstring $\mathrm{L}b$ whose locus is u, the contracted locus of $\mathrm{L}\alpha$. B is of dimension $l(u) \times l(u)$ because $\mathrm{L}b$ is an Lstring of that length. Before we discuss the second point, we need the following facts.

FACT 8. *Assume that $h = l(u)$, i.e., Lb and $L\alpha$ are Lstrings of the same length. Then u is the locus of $L\alpha$ implying that $Lb = L\alpha$, i.e., $A[i : i + h - 1, i : i + h - 1] = B$.*

Proof. The fact is proven by the definition of a contracted locus and locus of an Lstring. □

FACT 9. *Assume that u is not the locus of $L\alpha$, i.e., Lb is shorter than $L\alpha$, the Lstring corresponding to $A[i : i + h - 1, i : i + h - 1]$. $L\alpha$ is equal to the concatenation of Lb with the first $h - l(u)$ Lcharacters of the chunk on the edge (u, r'), i.e., $A[i : i + h - 1, i : i + h - 1]$ has B as a proper prefix and its remaining rows and columns correspond to the first $h - l(u)$ Lcharacters of the chunk on the edge (u, r').*

Proof. Since u is not the locus of $\mathrm{L}\alpha$, its extended locus r' is an offspring of u (by Lemma 5). Let $\mathrm{L}x$ be the Lstring of which r' is locus. $\mathrm{L}x$ is obtained by appending the chunk on the edge (u, r') to $\mathrm{L}b$. But $\mathrm{L}b$ is a proper Lprefix of $\mathrm{L}\alpha$, which is a proper Lprefix of $\mathrm{L}x$. Thus, $\mathrm{L}\alpha$ can be obtained from $\mathrm{L}b$ by appending to it the first $h - l(u)$ Lcharacters of the

chunk on the edge (u, r'), i.e., $A[i : i+h-1, i : i+h-1]$ has B as a proper prefix and its remaining rows and columns correspond to the first $h - l(u)$ Lcharacters of the chunk on the edge (u, r'). □

FACT 10. *Assume that there is no offspring r of u such that the first Lcharacter of the chunk on the edge (u, r) is $A[i+l(u), 1 : i+l(u)-1]A[1 : i+l(u), i+l(u)]$, i.e., the Lcharacter corresponding to $Lsuf_i(La\hat{\$})[l(u)+1]$. Then $Lb = Lhead_i$, i.e., $A[i : i+l_i-1, i : i+l_i-1] = B$.*

Proof. u is the contracted locus of $L\alpha$, an Lstring whose corresponding matrix is equal to $A[i : i+h-1, i : i+h-1]$. Thus the Lstring of which u is locus corresponds to a matrix equal to $A[i : i+l(u)-1, i : i+l(u)-1]$, which is a prefix of $A[i : n+1, i : n+1]$. So, Lb is an Lprefix of $Lsuf_i(La\hat{\$})$. Since none of the chunks on the edges outgoing u starts with $Lsuf_i(La\hat{\$})[l(u)+1]$, i.e., $A[i+l(u), 1 : i+l(u)-1]A[1 : i+l(u), i+l(u)]$, we have that Lb is the longest Lprefix that $Lsuf_i(La\hat{\$})$ has in common with $Lsuf_j(La\hat{\$})$, for $1 \leq j < n$ (by definition of LT_{i-1} and of trie over the alphabet $L\Sigma$). By definition of $Lhead_i$, we have that $Lb = Lhead_i$. □

For the sake of presentation on how we proceed for the second point, let us assume that f is the extended locus of $Lhead_i$ in LT_{i-1}. We anticipate that we will find the contracted locus of $Lhead_i$ by finding f and that we will establish the correctness of our approach while we present it. We also remark that in such a discussion we make critical use of Lemmas 4 and 6, as will be pointed out.

f must exist and it is in the subtree of LT_{i-1} rooted at u. Indeed, if $Lhead_i$ has a locus in LT_{i-1} then f is that locus (by definition of a locus and extended locus of an Lstring). In such a case, f is the node we are looking for and, by Lemma 6, it is in the subtree of LT_{i-1} rooted at u. If $Lhead_i$ does not have a locus in LT_{i-1}, its contracted locus is an internal node and f is one of its offsprings (by Lemma 4). In such a case, $parent(f)$ is the node we are looking for and, again by Lemma 6, f is in the subtree of LT_{i-1} rooted at u.

So, by finding f, we also find the contracted locus of $Lhead_i$ in LT_{i-1}. We start the search from u. We refer to $findpath(u, Lsuf_i(La\hat{\$}))$ as the operation that, starting from u, locates f in LT_{i-1}. It returns $parent(f)$ when f is not the locus of $Lhead_i$ in LT_{i-1}. Otherwise, it returns f. It can be described as follows.

Procedure *findpath*

- Select the only offspring r of u such that the first Lcharacter of the chunk on the edge (u, r) is $A[i+l(u), 1 : i+l(u)-1]A[1 : i+l(u), i+l(u)]$, i.e., the Lcharacter equal to $Lsuf_i(La\hat{\$})[l(u)+1]$. There are two cases to consider.
- **Case a.** No such offspring exists. By Fact 10, u is the locus of $Lhead_i$ and must be equal to f by definition of the extended locus of an Lstring. Exit.
- **Case b.** r exists. Rather than trying to be formal, we explain how we find f using Fig. 13. We need some preliminary observations. Fig. 13b gives the matrix $A[i : n+1, i : n+1]$ divided up into a prefix equal to B (by Facts 8 and 9, B is a prefix of $A[i : i+h-1, i : i+h-1]$) and chunks $\beta_1, \beta_2, \ldots, \beta_s, \ldots, \beta_q$. Such chunks correspond, in an obvious way, to the chunks $\beta'_1, \beta'_2, \ldots, \beta'_s, \ldots, \beta'_q$ on the edges of the path from u to f (shown in Fig. 13a). Note that β_e and β'_e, $1 \leq e \leq q$, have the same number of Lcharacters. Using such an observation and Fact 2 (applied to LT_{i-1}), one can easily prove that β_e and β'_e, $1 \leq e \leq q$, start with Lcharacters of the same length as strings in Σ^*. Therefore, by the concatenation rule for Lcharacters, the lth Lcharacter of β_e is, as a string of Σ^*, the same length as the lth Lcharacter of β'_e. Since β_e and β'_e can both be expressed as triples, we can use *compare* to compare their lth Lcharacters (recall its definition from §4). *findpath* skips the first $h - l(u)$ Lcharacters of the chunk on the edge (u, r), i.e., they are

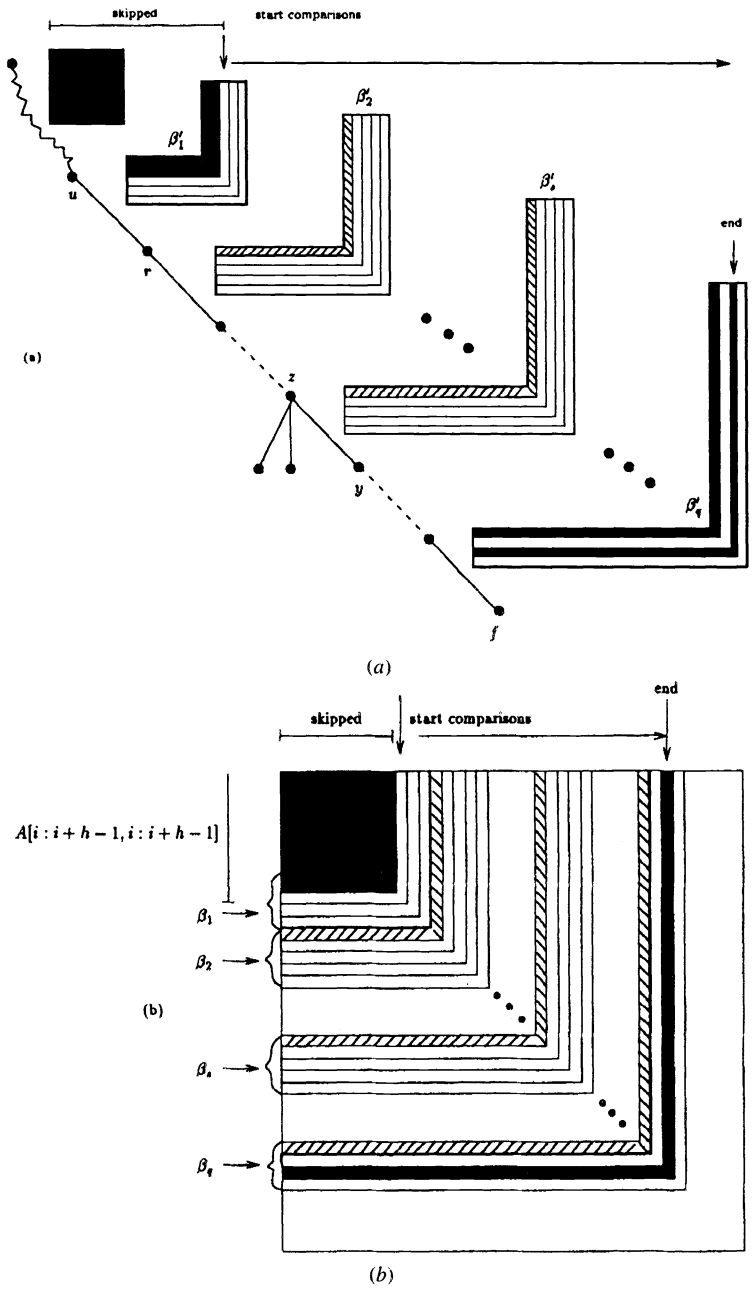


FIG. 13. (a) *The path from root* (LT_{i-1}) *to* f. (b) *The matrix* $A[i : n+1, i : n+1]$ *divided up into the matrix on the path from root* (LT_{i-1}) *to* u *and chunks corresponding to the ones on the edges of the path from* u *to* f.

not compared against the corresponding Lcharacters of the Lstring obtained from $A[i : n+1, i : n+1]$. By Facts 8 (case $h = l(u)$) and 9 (case $h > l(u)$), this amounts to skipping $A[i : i+h-1, i : i+h-1]$, the matrix corresponding to $L\alpha$. That is correct because by Lemma 5 such an Lstring is an Lprefix of $Lhead_i$. Starting with the Lcharacter of $A[i : n+1, i : n+1]$ pointed to by the arrow *start*, *findpath*

compares, one by one and from left to right, the Lcharacters of $A[i : n+1, i : n+1]$ with the corresponding ones on the edges of the path from u to f. A mismatch must eventually be found because the last Lcharacter of $A[i : n+1, i : n+1]$ is $\hat{\$}$ and it does not match any other Lcharacter. In Fig. 13, such a mismatch is between the Lcharacters pointed to by the arrow *end*. Notice that we perform $end - start + 1 = l_i - h + 1$ comparisons of Lcharacters in the chunks $\beta_1, \beta_2, \ldots, \beta_s, \ldots, \beta_q$ and $\beta'_1, \beta'_2, \ldots, \beta'_s, \ldots, \beta'_q$. *findpath* interleaves such comparisons with the "selection of the node to visit next" (recall that the path from u to f is not known to *findpath*). To this end, the Lcharacters in $A[i : n+1, i : n+1]$ between *start* and *end* with bars in them are special: they are the first Lcharacters of $\beta_2, \ldots, \beta_s, \ldots, \beta_q$ and are used by *findpath* to go from u to f. Indeed, assume that *findpath* has reached node z of the path from u to f. It selects the edge (z, y) whose label starts with an Lcharacter that is equal to the first Lcharacter of the chunk β_s of $A[i : n+1, i : n+1]$. The number of nodes visited is bounded by $l_i - h + 1$. Notice that, once we get to f, it is a simple matter to decide whether to return $parent(f)$ or f. We omit the details.

By Invariant 1, the edges outgoing each node of LT_{i-1} are sorted according to the first Lcharacter of the chunk outgoing each edge. So, the "selection of the node to visit next" can be done in $O(\log n)$ comparisons of Lcharacters. Therefore, the total time spent by *findpath* in comparisons of Lcharacters is bounded by $O((l_i - h + 1)\log n))$ (*findpath* visits at most $l_i - h + 1$ nodes and performs $l_i - h + 1$ comparisons of Lcharacters in the chunks $\beta_1, \beta_2, \ldots, \beta_s, \ldots, \beta_q$ and $\beta'_1, \beta'_2, \ldots, \beta'_s, \ldots, \beta'_q$). The remainder of the procedure takes $O((l_i - h + 1)\log n)$ additional time.

Recall that $h = \min(0, l_{i-1} - 1)$ and that *compare* takes constant time. For later reference, we summarize the above discussion in the following lemma.

LEMMA 9. *Starting from the contracted locus u of Lα, the procedure findpath correctly returns the contracted locus of Lhead$_i$ in LT$_{i-1}$. It takes* $O((l_i - h + 1)\log n) = O((l_i - \min(0, l_{i-1} - 1) + 1)\log n)$ *time.*

6.2.1. Point three. Recall the following facts. *findpath* returns a node $\hat{w}$ such that it is the locus of L*head*$_i$, if it exists in LT$_{i-1}$. Otherwise, $\hat{w}$ is its contracted locus. Moreover, $l(v)$ is known to **STI** by the invariant and Lα is of length $l(v) - 1$. Thus the length of L*head*$_i$ is easily computed knowing the length $l(v) - 1$ of Lα and the number of Lcharacter comparisons performed by *findpath*.

We refer to the procedure that actually transforms LT$_{i-1}$ into LT$_i$ as *updatetree*. It takes in input $\hat{w}$.

Procedure *updatetree*

- **Case a.** $\hat{w}$ is the locus of L*head*$_i$ in LT$_{i-1}$. A leaf g' is created as offspring of $\hat{w}$. It is labeled with i, since that leaf is the locus of the Lstring corresponding to $A[i : n+1, i : n+1]$, i.e., L*suf*$_i(La\hat{\$})$. The edge $(\hat{w}, g')$ is labeled with the triple $(l(\hat{w}) + 1, n + 1, i)$ that corresponds to the chunk obtained by deleting L*head*$_i$ from L*suf*$_i(La\hat{\$})$ ($l(\hat{w})$ is known by the invariant because $\hat{w}$ is a node of LT$_{i-1}$). The labeling is correct because $\hat{w}$ is locus of L*head*$_i$, g' is locus of L*suf*$_i(La\hat{\$})$, and the path from the root of LT$_i$ to g' must give that latter Lstring. $\hat{w}$ is made point to g'. Moreover, $SL(g)$ is set to g'. That is correct by the definition of a suffix link and the fact that g and g' are the locuses of L*suf*$_{i-1}(La\hat{\$})$ and L*suf*$_i(La\hat{\$})$, respectively.
- **Case b.** $\hat{w}$ is the contracted locus of L*head*$_i$ in LT$_{i-1}$, but that Lstring does not have a locus defined. Let f be its extended locus (it must exist by Lemma 4 and it is

known to $findpath$). Let (p_1, p_2, j) be the label on the edge $(\hat{w}, f) \in LT_{i-1}$. A new node w' is created as a locus of $Lhead_i$ by splitting the edge $(\hat{w}, f)$ (w' is made offspring of $\hat{w}$). The label on the edge $(\hat{w}, w')$ is set to $(p_1, p_1 + l_i - l(\hat{w}) - 1, j)$ and the one on the edge (w', f) to $(p_1 + l_i - l(\hat{w}), p_2, j)$. The labeling is correct. Indeed, the concatenation of the Lstring of which $\hat{w}$ is locus with the first $l_i - l(\hat{w})$ Lcharacters of the chunk on the former edge $(\hat{w}, f)$ is equal to $Lhead_i$. But the concatenation of the chunks on the edges $(\hat{w}, w')$ and (w', f) gives the chunk on the former edge $(\hat{w}, f)$. Thus the way $updatetree$ handles the split of the edge $(\hat{w}, f)$ does not change the set of Lstrings given by the paths of the Lsuffix tree before the split. $l(w')$ is correctly set to l_i since w' is the locus of $Lhead_i$. Then we proceed as in Case a, with $\hat{w}$ replaced by w'.

Note that the insertion of the new nodes in LT_{i-1} to transform it in LT_i can be done by a constant number of $link$ and cut operations (LT_{i-1} is a dynamic tree). Each such operation costs $O(\log n)$ time [25]. Moreover, the insertion of the new edges can be done in the same amount of time and in such a way that, for each node where they are inserted, the list of edges outgoing that node is still sorted according to the first Lcharacter of the chunk outgoing each edge. Using this fact and the invariant, we have that the edges outgoing each node of LT_i are still sorted according to the first Lcharacter of the chunk outgoing each edge. The remainder of the procedure takes constant time. For later reference, we summarize the above discussion into the following lemma.

LEMMA 10. *Assume that the invariant holds at the beginning of iteration i. Let w and g' be the locuses of $Lhead_i$ and $Lsuf_i(La\hat{\$})$ in LT_i. $updatetree$ correctly transforms LT_{i-1} into LT_i, computing, for w and g', the length of the Lstrings of which they are locuses and, for the new edges, their labels. It correctly sets $SL(g)$ to g' and w to point to g', a leaf in the subtree of LT_i rooted at w. It takes $O(\log n)$ time. Moreover, the edges outgoing each node of LT_i are sorted according to the first Lcharacter of the chunk outgoing each edge.*

6.3. Pseudocode, correctness, and time analysis. We now provide the pseudocode of **STI**, expressed in terms of the procedures given in the previous subsections. Then we prove its correctness and we analyze it.

Procedure STI—Pseudocode iteration i

1. If $v = root(LT_{i-1})$ then $u := root(LT_{i-1})$ and skip *rescanning*.
2. *rescanning*: $u := findclocus(v)$.
3. *scanning*: $\hat{w} := findpath(u, Lsuf_i(La\hat{\$}))$; $updatetree(\hat{w})$.

THEOREM 4. *Given an Lstring $La\hat{\$}$, corresponding to matrix $A[1 : n, 1 : n]\hat{\$}$, and an initial Lsuffix tree LT_0 of one node,* **STI** *correctly builds the Lsuffix tree for $La\hat{\$}$ by inserting, from longest to shortest, the Lsuffixes of $La\hat{\$}$ into LT_0. It takes $O(n \log n)$ time.*

Proof. The proof of correctness is by induction. During iteration i, **STI** inserts $Lsuf_i(La\hat{\$})$, the Lstring corresponding to $A[i : n + 1, i : n + 1]$, into LT_{i-1}, the Lsuffix tree containing the Lstrings corresponding to the first $i - 1$ suffixes of A. Our inductive hypothesis is that at the beginning of the ith iteration, LT_{i-1} is known and that Invariant 1 holds. We show that during the ith iteration, procedure **STI** correctly transforms LT_{i-1} into LT_i and maintains the invariant. Therefore, our inductive hypothesis will be satisfied at the beginning of iteration $i + 1$. That will prove that **STI** correctly builds LT_n, the Lsuffix tree for $La\hat{\$}$.

Note that for $i = 1$, $LT_{i-1} = LT_0$ is known. It consists of one node v (the root) which is the locus of $Lhead_0$, the empty Lstring. v is also a leaf locus of $Lsuf_0(La\hat{\$})$, because that Lstring is empty. $l(v) = 0$ and there are no internal nodes. So, the invariant is satisfied for $i = 1$. Let us assume that it holds for $i > 1$; we show that it holds for $i + 1$. Recall that $L\alpha$ is

$\mathrm{L}suf_2(\mathrm{L}head_{i-1})$. Let w and g' be the locuses of $\mathrm{L}head_i$ and $\mathrm{L}suf_i(\mathrm{L}a\hat{\$})$ in $\mathrm{L}T_i$, respectively. We consider two cases corresponding to whether or not rescanning is performed.

When the contracted locus u of $\mathrm{L}\alpha$ is $root(LT_{i-1})$, **STI** skips *rescanning*. By Lemma 9, $findpath$ correctly finds the contracted locus $\hat{w} \in LT_{i-1}$ of $\mathrm{L}head_i$. By Lemma 10, $updatetree$ correctly transforms $\mathrm{L}T_{i-1}$ into $\mathrm{L}T_i$, computing, for w and g', the length of the Lstrings of which such nodes are locuses. Since the length of the Lstrings of which the nodes of $\mathrm{L}T_{i-1}$ are locuses is known by the invariant, we know $l(z)$, for all $z \in LT_i$. $findpath$ also correctly computes $SL(g)$, where g is the only leaf in $\mathrm{L}T_{i-1}$ that did not have a value of SL defined (by the invariant). Therefore, g' is the only leaf in $\mathrm{L}T_i$ that does not have a value of SL defined. w is made to point to g', a leaf in the subtree of $\mathrm{L}T_i$ rooted at w. Since w is possibly the only internal node of $\mathrm{L}T_i$ not in $\mathrm{L}T_{i-1}$ and the invariant holds, we have that each internal node $c \in LT_i$ has a pointer to a leaf (arbitrarily chosen) in the subtree of $\mathrm{L}T_i$ rooted at c. Finally, w and g', the locuses of $\mathrm{L}head_i$ and $\mathrm{L}suf_i(\mathrm{L}a\hat{\$})$ in $\mathrm{L}T_i$ are known to the algorithm. Thus the invariant is satisfied.

When the contracted locus u of $\mathrm{L}\alpha$ is not $root(LT_{i-1})$, **STI** finds that node by calling $findclocus$, which correctly returns u by Lemma 7. The proof that our inductive hypothesis holds for $i+1$ also in this case is now as in the preceding case.

As for the time analysis, note that there can be at most n calls to the procedures $findclocus$, $findpath$, and $updatetree$. The cost of the ith iteration is given by adding the bounds in Lemmas 8, 9, and 10. Thus it is bounded by $O((l_i - \min(0, l_{i-1} - 1) + 1) \log n)$ time. Adding over all iterations, we get the claimed bounds because $l_1 \leq n+1$ and $l_{n+1} = 0$ ($\hat{\$}$ does not match anything). □

7. Construction of the Lsuffix tree for matrix A. Given an $n \times n$ matrix A, recall from §3.2 that the Lsuffix tree for A is the Lsuffix tree for the set of Lstrings corresponding to $A_d\hat{\$}_d$, $0 \leq |d| < n$, i.e., $D = \{\mathrm{L}a_d\hat{\$}_d,\ 0 \leq |d| < n\}$. We build such a data structure by "reducing" the problem to the one of building the Lsuffix tree of a single Lstring (which we know how to do with the algorithm of the previous section). Indeed, let B be the matrix obtained from $A_d\hat{\$}_d$, $0 \leq |d| < n$, as shown in Fig. 14. Let $\hat{n} = n^2 + 2n - 1$ and $\mathrm{L}T_b$ be the Lsuffix tree for $\mathrm{L}b\hat{\$}$. In the remainder of this section, let q be the leaf of LT_b that is the locus of the Lstring corresponding to $B[\hat{n}+1, \hat{n}+1] = \$$, i.e., the leaf that is the locus of the Lstring corresponding to the last suffix of $B\hat{\$}$. We need the following lemma.

LEMMA 11. *The Lsuffix tree LT_D for the Lstrings in $D = \{\mathrm{L}a_d\hat{\$}_d,\ 0 \leq |d| < n\}$ is isomorphic to $LT_b - \{q\}$, where LT_b is the Lsuffix tree for Lstring $\mathrm{L}b\hat{\$}$ (the Lstring corresponding to matrix $B\hat{\$}$).*

Proof. We establish a one-to-one correspondence between the nodes of $\mathrm{L}T_D$ and the ones of $\mathrm{L}T_b - \{q\}$.

We first establish a one-to-one correspondence between the leaves of $\mathrm{L}T_D$ and the ones of $\mathrm{L}T_b - \{q\}$. By definition of an Lsuffix tree of an Lstring (set of Lstrings, respectively), there is a one-to-one correspondence between the leaves of LT_b (LT_D, respectively) and the suffixes of $B\hat{\$}$ ($A_d\hat{\$}_d$, $0 \leq |d| < n$, respectively). But there is a one-to-one correspondence between the suffixes of B and the ones of $A_d\hat{\$}_d$ (as shown in Fig. 14), which proves our claim.

We now show that there is a one-to-one correspondence between the internal nodes of LT_D and the ones of LT_b. Let C be a square submatrix of B along its main diagonal and such that C has no rows and columns of special characters $\$_d$. Given the correspondence between suffixes of B and suffixes of A_d, $0 \leq |d| < n$, any such submatrix C of B is a prefix of some suffix of B if and only if C is a prefix of some suffix of A_d, $0 \leq |d| < n$. Using this fact, the fact that $\hat{\$}_d$, $0 \leq |d| < n$, does not match anything, and the correspondence between square matrices and Lstrings, it is straightforward to prove that an Lstring has an internal node as a

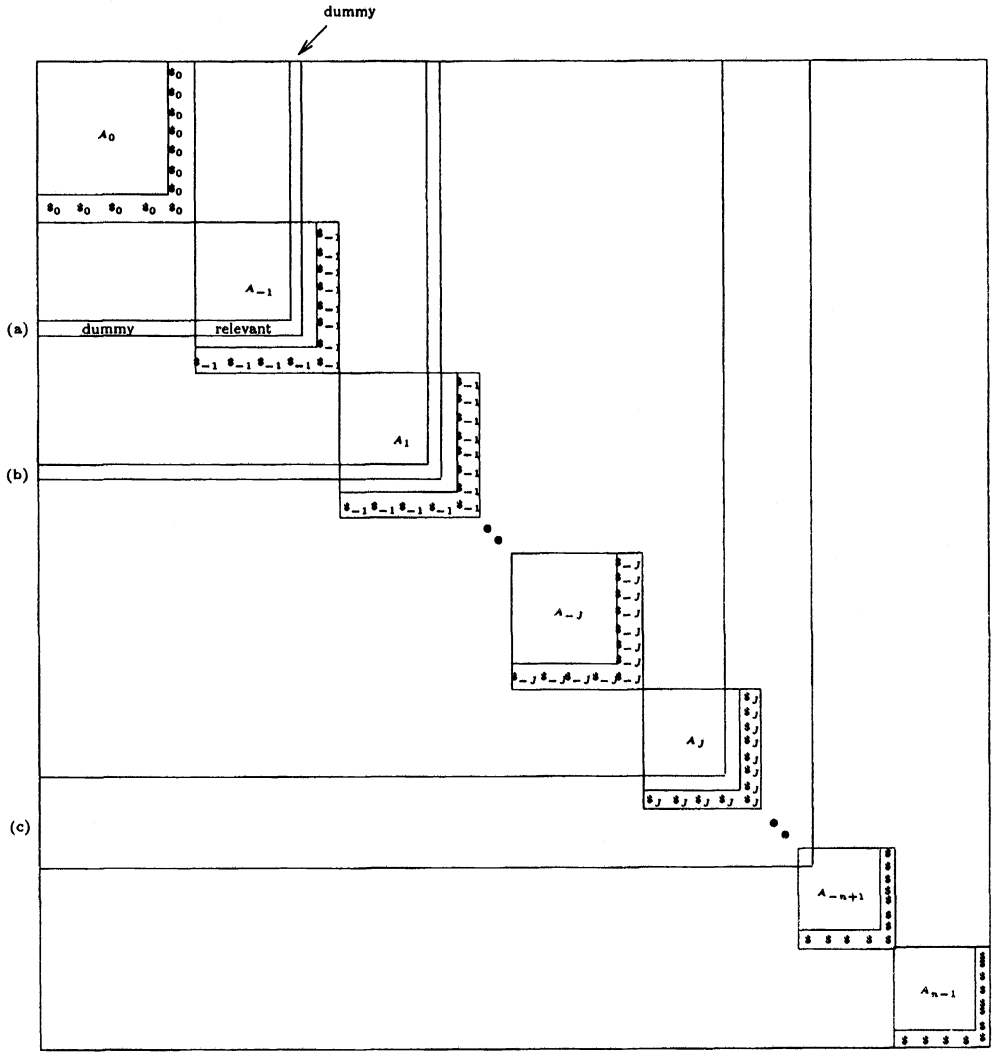


FIG. 14. *Matrix B corresponding to* $A_d\hat{\$}_d$, $0 \leq |d| < n$. *Entries of B outside* $A_d\hat{\$}_d$, $0 \leq |d| < n$, *have dummy symbols (not shown).*

locus in LT_b if and only if it has an internal node as a locus in LT_D. Since the internal nodes of both trees are locuses of Lstrings, we have a one-to-one correspondence between the internal nodes of LT_D and the ones of LT_b. □

Before we describe how to obtain LT_D from LT_b, we need a few preliminary observations. Note that B is an $\hat{n} \times \hat{n}$ matrix such that each row (column, respectively) is obtained from a given subrow (subcolumn, respectively) of A by adding dummy symbols to the beginning and end of it. Thus, there is no need to build B explicitly. Moreover, we point out that each chunk (and in particular Lcharacters) of $Lb\hat{\$}$ and its Lsuffixes is made up of a *dummy part* covering subrows and subcolumns of B with dummy characters in them and a *relevant part* covering subrows and subcolumns of B that correspond to subrows and subcolumns of matrices $A_d\hat{\$}$, $0 \leq |d| < n$ (for an example, see chunk (a) in Fig. 14).

We can compare Lcharacters from Lb and any of its Lsuffixes as follows. We preprocess the matrix A as in §4. By Lemma 1, that takes $O(n^2 \log |\Sigma|)$ time. Then, given two Lcharacters from triples of chunks of Lb or of its Lsuffixes, we take only the starting points in B of an occurrence of the relevant parts of those Lcharacters. Using the correspondence (shown in Fig. 14) between the matrices $A_d\hat{\$}$, $0 \leq |d| < n$, and the suffixes of B, such starting points can be translated, in constant time, into the starting points in A of an occurrence of two Lcharacters of $La_d\hat{\$}_d$, $0 \leq |d| < n$, or one of its Lsuffixes. Then we can use *compare* as described in §4. The bounds in Lemma 1 still hold.

We can build the Lsuffix tree LT_D of matrix A by building the Lsuffix tree LT_b for the Lstring $Lb\hat{\$}$ (corresponding to matrix $B\hat{\$}$). By Lemma 11, LT_D is isomorphic to $LT_b - \{q\}$. Then we have to transform $LT_b - \{q\}$ into LT_D by a suitable transformation of labels (which will be outlined shortly). We proceed as follows. We use the procedure of §§5 and 6 to build LT_b with the following changes. Any time we need an Lcharacter from Lb or one of its Lsuffixes, we recover its relevant part in matrix A through the correspondence (shown in Fig. 14) between the matrices $A_d\hat{\$}$, $0 \leq |d| < n$, and the suffixes of B. Such a translation takes constant time. Moreover, when we need to compare Lcharacters from $Lb\hat{\$}$ or any of its Lsuffixes, we use *compare* as described above. Notice that it still takes constant time. Using the fact that $\hat{n} = O(n^2)$ and Theorem 4, we can build LT_b in $O(n^2 \log n)$ time.

Once we have LT_b, we delete leaf q and translate the triples representing chunks of Lb and its Lsuffixes into quadruples representing chunks of Lstrings $La_d\hat{\$}_d$ and their Lsuffixes, $0 \leq |d| < n$. Let (p, q, l) be a triple on an edge (u, v) of $LT_b - \{q\}$. It represents chunk $Lsuf_l(Lb\hat{\$})$. There are two cases to consider: one in which the relevant part of (p, q, l) is fully within a suffix of some A_d (see chunk (b) in Fig. 14) and the other in which it is not (see chunk (c) in Fig. 14). For the case depicted in Fig. 14b, we translate the triple (p, q, l) into a quadruple that represents only the relevant part of $Lsuf_l(Lb\hat{\$})$ and therefore corresponds to the correct chunk of Lstrings $La_d\hat{\$}_d$ and their Lsuffixes, $0 \leq |d| < n$, that is to be placed on the edge (u, v) of LT_D. In the other case, $v \in LT_b$ is a leaf since $\$_j$ occurs only once in B and $\$_j$ is in the relevant part of (p, q, l) (see Fig. 14c). We translate such a triple into the correct quadruple to be placed on the edge $(u, v) \in LT_D$ by keeping only the relevant part of (p, q, l) (which is above the subrow and to the right of the subcolumn of $\$_j$'s in B). In both cases, this transformation takes constant time. Since the size of LT_b is $O(\hat{n})$ (by Fact 4) and since $\hat{n} = O(n^2)$, we get that we can transform $LT_b - \{q\}$ into LT_D in $O(n^2)$ time. Moreover, the transformation is such that the edges outgoing each node of LT_D are sorted according to the first Lcharacter on the chunk of each edge. Recalling that preprocessing of the matrix A takes $O(n^2 \log |\Sigma|)$ time and that LT_b can be built in $O(n^2 \log n)$ time, we get the following theorem.

THEOREM 5. *Given an $n \times n$ matrix A, we can build the Lsuffix tree for matrix A in $O(n^2(\log n + \log |\Sigma|))$ time. Moreover, the edges outgoing each node of LT_D are sorted according to the first Lcharacter on the chunk of each edge.*

8. Refining the Lsuffix tree for matrix A. As pointed out at the end of §3.2.3, the Lsuffix tree LT_D for matrix A can be used to find all occurrences of a matrix PAT into A. However, the outdegree of nodes in LT_D may be large, resulting in a slow-down in the search time. We can avoid such a slow-down by reducing the outdegree of the nodes in LT_D, i.e., by refining it. Indeed, the *refinement* of the Lsuffix tree for A, referred to as the RLsuffix tree for A and denoted RLT_D, is a compacted trie defined over Σ. Thus the outdegree of each node of the Lsuffix tree is at most Σ and we can "jump" from one node to another in $O(\log |\Sigma|)$ time. Intuitively, it represents the same information as LT_D but in a different format. Indeed, LT_D represents Lstrings, which are "strings" defined over an alphabet of Lcharacters. Since

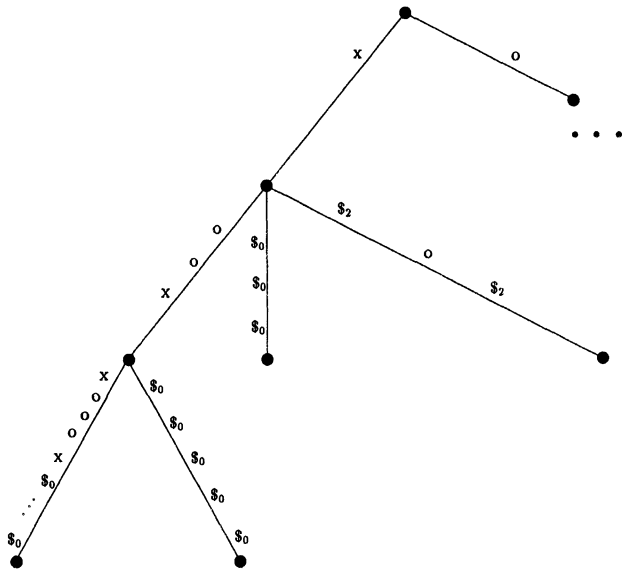


FIG. 15. *The refinement of the Lsuffix tree of Fig.* 10. *Only the part corresponding to the suffixes of* A_0 *and* A_2 *are shown.*

Lcharacters are strings over Σ, we can represent Lstrings as strings in Σ^*. In fact, RLT_D represents the same Lstrings as LT_D but as strings in Σ^*. (See Fig. 15 for an example.) Let $\leq$ denote the lexicographic order relation for strings of Σ^*.

8.1. Construction of RLT_D: High-level description. We give a constructive definition of RLT_D by describing, at a high level, how it is obtained from LT_D. In what follows, for a given internal node $v \in LT_D$, let $off(v)$ be the number of offsprings of v in LT_D and let $w_1 < \cdots < w_{off(v)}$ be the list of its offsprings sorted according to the first Lcharacter of the chunks on the corresponding edges. Such a list can be easily obtained from LT_D in $O(off(v))$ time, since we keep the edges outgoing v in the order $w_1 < \cdots < w_{off(v)}$ (see Theorem 5). Let β_i denote the string that corresponds to the chunk on the edge (v, w_i), $1 \leq i \leq off(v)$, when we write it down as a string in Σ^*. For each internal node $v \in LT_D$, we build a compacted trie $PT(v)$ defined over Σ. It represents, as strings in Σ^*, the chunks on the edges outgoing v, i.e., it represents $\beta_1, \ldots, \beta_{off(v)}$. So, each edge of $PT(v)$ has a string assigned to it, which is a substring of some β_i. Since each chunk on the edges outgoing v starts with a different Lcharacter (by definition of Lsuffix tree), no β_i can be a prefix of any other β_j. Thus there is a one-to-one correspondence between the leaves of $PT(v)$ and the offsprings of v. We make the root of $PT(v)$ correspond to v. Once the $PT(v)$'s have been computed for each internal node $v \in LT_D$, we glue them together according to the parent–offspring relation in LT_D. That is, for each edge $(v, f') \in LT_D$, the root of $PT(f')$ is coalesced with the leaf corresponding to it in $PT(v)$. The resulting tree is RLT_D.

Note that with such a construction there is no loss of information. That is, RLT_D represents the same information as LT_D, but in a different format. Moreover, there is a one-to-one correspondence between the leaves of LT_D and the ones of RLT_D. Indeed, the leaf $f' \in LT_D$ labeled (l, i) and such that the path from the root of LT_D to f' spells out the Lstring $Lsuf_l(La_i\hat{\$}_i)$ corresponds to the leaf $f \in RLT_D$ such that the path from the root of RLT_D to f spells out $Lsuf_l(La_i\hat{\$}_i)$ as a string of Σ^*. The converse is also true. We assign the label (l, i) to f. Note that RLT_D has $O(n^2)$ leaves.

In what follows we bound the size of RLT_D. We start by describing how the strings on its edges are represented. This involves discussing the representation of strings on the edges of a single $PT(v)$. Given a string γ on an edge of $PT(v)$, we represent it in constant space with a sixtuple (p, q, l, i, bgn, end) which means the following. Let β be the string obtained by writing down the chunk $\mathrm{L}suf_l(La_i\hat{\$}_i)[p, q]$ represented by quadruple (p, q, l, i) (which is on some edge $(v, w_j) \in LT_D$). Then $\gamma = \beta[bgn, end]$. We need the following fact, which will be useful in the next section.

FACT 11. *Given the sixtuple (p, q, l, i, bgn, end), let γ be the string it represents. A prefix $\hat{\gamma}$ of γ can be recovered from the sixtuple in $O(|\hat{\gamma}|)$ time.*

Proof. (p, q, l, i, bgn, end) encodes $\gamma = \beta[bgn, end]$, where β corresponds to the chunk $\mathrm{L}suf_l(La_i\hat{\$}_i)[p, q]$ as a string of Σ^*. Using the definition of chunk, we know that the tth, $p \leq t \leq q$, Lcharacter of that chunk has length $2t - 1$ as a string of Σ^*. Let k be the maximal integer such that $\Sigma_{i=0}^{k}(2(p + i) - 1) = k^2 + 2p(k + 1) - 1 < bgn$. It can be found in constant time. In order to get to the starting point of γ and $\hat{\gamma}$ in β, we need to skip the first $k + 1$ Lcharacters of $\mathrm{L}suf_l(La_i\hat{\$}_i)[p, q]$. Skipping the first $bgn - 1$ characters, we take the subsequent $end - bgn + 1$ characters out of the ones that compose the Lcharacters in the chunk $\mathrm{L}suf_l(La_i\hat{\$}_i)[p + k + 1, q]$. They correspond to $\hat{\gamma}$. By Theorem 3, we can access the starting points of an occurrence of any of those Lcharacters in A in constant time. So, using the correspondence between Lcharacter and subrows and subcolumns of A, we can get $\hat{\gamma}$ in time linear in its size. □

FACT 12. *The size of $PT(v)$, i.e., number of nodes, edges, and labels, is $O(off(v))$.*

Proof. $PT(v)$ is a compacted trie over the alphabet Σ that has $O(off(v))$ leaves. By the definition of a compacted trie [20], each internal node of $PT(v)$, except possibly the root, has outdegree at least two. Thus the number of nodes, edges, and labels (one per edge) is bounded by $O(off(v))$. □

LEMMA 12. *Given a node g in RLT_D, let lf be the number of leaves in the subtree T_g of RLT_D rooted at g. The number of nodes in T_g is $O(lf)$.*

Proof. Since each $PT(v)$, $v \in LT_D$ and internal node, is a compacted trie, only its root can have outdegree one. Moreover, since $PT(v)$ represents at least two different strings (no β_i is prefix of any other β_j and $off(v) \geq 2$), each leaf of $PT(v)$ must descend from a parent of outdegree at least two. Since RLT_D is obtained by gluing all the $PT(v)$'s together, for each internal node $v \in LT_D$, the above properties of $PT(v)$ imply the following properties for RLT_D: (a) the nodes of outdegree one in RLT_D must be offsprings of nodes of outdegree at least two; (b) all leaves of RLT_D must be offsprings of internal nodes of outdegree at least two.

Consider a tree T'_g obtained from T_g by "contracting" the internal nodes of outdegree one into their offsprings. Note that because of property (b) of RLT_D, T'_g has the same number of leaves of T_g and, because of property (a) of RLT_D, it is at least half the size of T_g. Since each node in T'_g has outdegree at least two, its size is $O(lf)$, which proves the claimed bound on the size of T_g. □

LEMMA 13. *The size of RLT_D, i.e., number of nodes, edges, and labels, is $O(n^2)$.*

Proof. The lemma follows from Lemma 12 (applied to the root of RLT_D) and the fact that RLT_D has $O(n^2)$ leaves. □

THEOREM 6. *Let B be a square matrix and let Lb be the corresponding Lstring. B is submatrix of A if and only if the string corresponding to Lb has an extended locus w in RLT_D. Moreover, let F be any square submatrix of A having B as a prefix and let Lf be the Lstring corresponding to F. The extended locus of the string corresponding to Lf in RLT_D is a descendant of w.*

Proof. Given the way in which we obtain RLT_D from LT_D, the theorem is a simple variation of Theorem 2. The proof is omitted. □

8.2. Construction of RLT_D: Low-level description. Let us assume that LT_D has been built and let us assume that the matrix A has been preprocessed so that *compare* takes constant time. The main point in the construction of RLT_D is the computation of $PT(v)$, $v \in LT_D$, which we now present.

Recall that $w_1 < \cdots < w_{off(v)}$ is the list of offsprings of v in LT_D sorted according to the first Lcharacter of the chunks on the corresponding edges and that β_i denotes the string that corresponds to the chunk on the edge (v, w_i), $1 \le i \le off(v)$, when we write it down as a string in Σ^*. Note that, for $i < j$, $\beta_i < \beta_j$ because $w_i < w_j$. Moreover, the longest prefix that β_i and β_j have in common is "confined" to the first Lcharacter of the chunks on the edges (v, w_i) and (v, w_j) (no two such chunks can start with the same Lcharacter, by definition of an Lsuffix tree). Since we have pointers to such Lcharacters (the quadruples representing the chunks), we can compute the length of such a prefix by using *compare* (recall its definition from §4). Note also that given a quadruple representing a chunk, we can compute the length of the string corresponding to the chunk in constant time. Thus, we can assume that the lengths of $\beta_1, \ldots, \beta_{off(v)}$ are known.

We build $PT(v)$ incrementally by inserting into it the path representing β_i, once the path representing β_r is already there, for $1 \le r < i$. Assume that we have $PT_{i-1}(v)$, the compacted trie that represents $\beta_1 < \cdots < \beta_{i-1}$, and assume that the following invariant holds.

INVARIANT 2. *The edges outgoing any node of $PT_{i-1}(v)$ are sorted according to the first character of the string on each edge. The array $PATH$ stores the nodes of $\hat{p}$, where $\hat{p}$ is the path representing β_{i-1} in $PT_{i-1}(v)$. Moreover, the nodes in $PATH$ are sorted in increasing order of the length of the strings of which they are locuses. For each node $c \in \hat{p}$, the length $len(c)$ of the string of which c is a locus is known.*

Note that for $i = 2$, $PT_1(v)$ is an edge representing β_1 and $\hat{p}$ is that edge (it can be stored in $PATH$ in constant time). Moreover, since we know the length of β_1 and $root(PT_{i-1}(v))$ is the locus of the empty string, we also know the length of the strings of which the nodes in $\hat{p}$ are locuses. Since $PT_1(v)$ has only one edge, all edges outgoing the same internal node are sorted according to the first character of the strings on those edges. Thus Invariant 2 is satisfied for $i = 2$. Now we show how to transform $PT_{i-1}(v)$ into $PT_i(v)$ and how to maintain Invariant 2 for $i \ge 2$. We do not actually provide all details, since many of them are quite standard operations in manipulating tries.

Transformation of $PT_{i-1}(v)$ into $PT_i(v)$. Let len be the length of the longest prefix common to β_{i-1} and β_i. We can compute it in constant time using *compare*, as already discussed. Since $\beta_r < \beta_{i-1}$, for $r < i - 1$, no β_r can have a prefix in common with β_i of length longer than len. This implies that we can create the path p representing β_i by making it "share" $\beta_i[1, len]$ with $\hat{p}$. Thus p is obtained from $\hat{p}$ by creating, if needed, a locus u for $\beta_i[1, len]$ on $\hat{p}$ and then, as an offspring of u, a leaf f representing $\beta_i[len + 1, |\beta_i|]$. This is done as follows. We have the nodes of $\hat{p}$ in $PATH$, sorted in increasing order of the length of the strings of which they are locuses. We find where to create u on $\hat{p}$ by finding, through a binary search on $PATH$, the maximal node c for which $len(c) \le len$. If $len(c) = len$ then $u = c$. Otherwise u is created by splitting the edge $(c, c') \in PT_{i-1}(v)$ whose label starts with $\beta_i[len(c) + 1]$ (u becomes an offspring of c and the parent of c' in $PT_i(v)$). Moreover, $len(u) = len$ and $len(f) = |\beta_i|$. Note that the split of the edge $(c, c') \in PT_{i-1}$ causes the split of the label on that edge into two. That can be done in constant time. We omit the details. Moreover, the sixtuple that has to label the edge $(u, f) \in PT_i(v)$ can be obtained in constant time from the quadruple labeling the edge $(v, w_i) \in LT_D$ and len. Again, we omit the details. We also remark that the edge (u, f) and possibly the edges (c, u) and (u, c') can be placed

in such a way that all edges outgoing the same internal node of $PT_i(v)$ are sorted according to the first character of the strings on those edges. Again, we omit the details. The above discussion proves that part of Invariant 2 will hold for $PT_i(v)$.

Update of $PATH$. We update $PATH$ to contain p by throwing out all descendants of c (which follow it in $PATH$) and appending to it u and the new leaf f. The descendants of c can be deleted in constant time by moving the right boundary of $PATH$ so that c is the last element of $PATH$. The update of $PATH$ establishes that the remainder of Invariant 2 will hold for $PT_i(v)$.

LEMMA 14. *$PT(v)$ can be correctly built in $O(off(v)(\log|\Sigma| + \log n))$ time and in such a way that edges outgoing any node of $PT(v)$ are sorted according to the first character of the string on each edge.*

Proof. The proof of correctness is by induction. It is given by the discussion preceding the lemma. The most expensive step in the transformation of $PT_{i-1}(v)$ into $PT_i(v)$ is a binary search on $PATH$. The rest of the work requires constant time (including the update of $PATH$ and calls to *compare*) or $O(\log|\Sigma|)$ time (to insert the new edges so that the list of edges outgoing any node of $PT_i(v)$ is sorted according to the first character on those edges). Since the size of $PT(v)$ is bounded by $O(off(v))$ (by Fact 12) and $|PT_j(v)| \leq |PT(v)|$, $j = 1 \ldots off(v)$, $PATH$ has at most $off(v) \leq n^2$ nodes in it. Thus a binary search takes $O(off(v)\log n)$ time. Summing the work for $i = 2, \ldots, off(v)$, we obtain the claimed bound. □

THEOREM 7. *RLT_D can be built in $O(n^2(\log|\Sigma| + \log n))$ time and in such a way that the edges outgoing any node of RLT_D are sorted according to the first character of the string on each edge.*

Proof. For each internal node $v \in LT_D$, the edges outgoing any node of $PT(v)$ are sorted according to the first character of the string on each edge (by Lemma 14). RLT_D is obtained by gluing those trees together, so it has the same property. The work required to build $PT(v)$ for each internal node $v \in LT_D$ is bounded in Lemma 14. Since the sum of the $off(v)$'s over all internal nodes v of LT_D is bounded by $O(n^2)$, we get that the total work to build the $PT(v)$'s is bounded by $O(n^2)$ calls to *compare* and $O(n^2(\log|\Sigma| + \log n))$ additional time. The bound of the theorem follows by noting that *compare* takes a constant time per call and that gluing the $PT(v)$'s together can be done in $O(n^2)$ time. □

9. Application one: Pattern retrieval. In this section we apply the Lsuffix tree and its refinement to derive efficient algorithms for the pattern retrieval problem. In order to simplify the presentation, we limit our discussion to the case in which the library S has only one text, an $n \times n$ matrix $TEXT$. The results generalize easily to a set of texts.

9.1. Pattern retrieval through an index. We are interested in building an index representing all square submatrices of $TEXT$ (preprocessing step). Then, given an $m \times m$ matrix PAT, we want to use the index to find all occurrences of PAT in $TEXT$. Let $TEXT_d$ be the square submatrix of $TEXT$ whose main diagonal is the dth diagonal of $TEXT$, $0 \leq |d| \leq n-1$, and let Lt_d denote the Lstring corresponding to $TEXT_d$, $0 \leq |d| \leq n-1$. Moreover, let $D = \{Lt_d\hat{\$}_d, 0 \leq |d| < n\}$.

The preprocessing step consists of building the Lsuffix tree LT_D for matrix $TEXT$ and then refining it. By Theorems 5 and 7 that will take $O(n^2(\log|\Sigma| + \log n))$ time.

Let $Lpat$ be the Lstring corresponding to matrix PAT and let pat be $Lpat$, when we write it down as a string of Σ^*.

THEOREM 8. *PAT occurs in $TEXT$ if and only if pat has an extended locus u in RLT_D. Moreover, all such occurrences are prefixes of suffixes of $TEXT_d$, $0 \leq |d| < n$, corresponding to the leaves of the subtree of RLT_D rooted at u.*

Proof. PAT occurs in $TEXT$ if and only if it is a square submatrix of $TEXT$ and, by Theorem 6, if and only if *pat* has an extended locus u in RLT_D. That proves the first part of the theorem. For the second part, note that PAT occurs in $TEXT$ if and only if it is the prefix of some suffix of $TEXT_d$. Let C be one of these suffixes, Lc be the Lstring corresponding to it, and c be the string obtained from Lc when we consider Lcharacters as strings. By Theorem 6, c has an extended locus in RLT_D. That extended locus must be a leaf f (it follows from the way we build RLT_D from LT_D and the fact that the extended locus of Lc in LT_D is a leaf). Again by Theorem 6, f must be in the subtree of RLT_D rooted at u (PAT is a prefix of C). □

Given PAT, we can construct *pat* in $O(m^2)$ time using the correspondence between matrices and Lstrings. Then the search for the extended locus of *pat* in RLT_D is a nearly standard search for the extended locus of a string in a compacted trie defined over Σ. However, we have to be careful with the fact that there are sixtuples on the edges of RLT_D, rather than actual strings. We give details only on how to traverse one edge of RLT_D guided by *pat*.

Assume that we have reached a node $g \in RLT_D$, locus of $pat[1, s]$. We select the edge to traverse the next by finding the edge (g, h) whose label starts with $pat[s+1]$. That can be done in $O(\log|\Sigma|)$ time because, by Theorem 7, the edges outgoing any node of RLT_D are sorted according to the first character of the string on each edge. Let (p, q, l, i, bgn, end) be the label on the edge (g, h) and let γ be the string it represents. Let $l = \min(m^2 - s, end - bgn + 1)$. We extract from the sixtuple a prefix $\hat{\gamma}$ of γ of length l. By Fact 11, this can be done in $O(l)$ time. Then we compare $\hat{\gamma}$ with $pat[s+1, s+l]$. This takes $O(l)$ time. If they are equal, we go on with the edge traversal starting from h and guided by $pat[s+l+1, m^2]$. Otherwise, there is no extended locus of *pat* in RLT_D.

Such an edge traversal costs $O(l + \log|\Sigma|)$ time. Since we never back up in reading the input, the whole time bound for the search procedure is bounded by $O(m^2 \log|\Sigma|)$ time.

Once we have reached the extended locus u of *pat* in RLT_D, we know, by Theorem 8, that the *occ* occurrences of PAT in $TEXT$ are given by the labels on the *occ* leaves of the subtree T_u of RLT_D rooted at u. By Lemma 12, the size of T_u is $O(occ)$, so we can visit its leaves, starting from u, in $O(occ)$ time. We summarize the above discussion in the following theorem.

THEOREM 9. *Finding all occurrences of matrix PAT in TEXT takes* $O(m^2 \log|\Sigma| + occ)$ *time, where occ is the number of occurrences of PAT in TEXT.*

9.2. Pattern retrieval through a set of table. Here we outline how to convert the refined Lsuffix tree RLT_D for matrix $TEXT$ into a set of tables, which can then be used to perform pattern matching. We assume that the reader is throughly familiar with the notation and approach of Manber and Myers to build suffix arrays for strings [21], which we briefly summarize.

Given a text string, Manber and Myers define three tables that contain lexicographic information about all suffixes of the text string and show how to compute such tables using a doubling technique. Using the tables, they devise a simple binary search procedure that finds all occurrences of a given pattern string in the text string. Such a procedure finds the set of suffixes of text that have the pattern as a prefix. The time performance of the search procedure depends on the size N of the tables and the length m of the pattern string: it is $O(m + \log N + occ)$, where *occ* is the number of occurrences of the pattern in the text.

Let $\alpha_{l,d}$ be the string corresponding to $Lsuf_l(Lt_d)$, $0 \le |d| < n$ and $1 \le l \le n - |d|$. Recall from §8 that the concatenation of the labels on the path from the root of RLT_D to a leaf labeled (l, d) gives $\alpha_{l,d}$. Here we also define three tables, which are the same as the ones of Manber and Myers, except that they contain lexicographic information about the n^2 strings $\alpha_{l,d}$. However, we do not use the doubling technique of Manber and Myers since we obtain

such tables directly from RLT_D. Then the procedure that searches for matrix PAT in $TEXT$ is the one of Manber and Myers, where the pattern string is pat. Thus such a procedure will find the set of $\alpha_{l,d}$, $0 \leq |d| < n$ and $1 \leq l \leq n - |d|$ that has pat as prefix. Because of the one-to-one correspondence between the α's, the Lsuffixes of Lt_d and the suffixes of $TEXT_d$, this is equivalent to finding all suffixes of $TEXT_d$, $0 \leq |d| < n$, that have PAT as a prefix (a proof of this fact is implicit in Theorem 8). In what follows, we give only an outline of our construction since the rationale behind it is as in [21].

Let us assume that RLT_D has been computed. By Theorem 7, it can be built in $O(n^2(\log|\Sigma| + \log n))$ time. As in [21], let us define an array $POS[0, n^2 - 1]$ of pairs such that $\alpha_{POS[k]} \leq \alpha_{POS[j]}$ for $k \leq j$. We use RLT_D to compute POS. We can compute POS by traversing RLT_D alphabetically since such visit will give us the α's sorted lexicographically. By Theorem 7, the edges outgoing any node of RLT_D are sorted according to the first character of the string on each edge, so such alphabetic visit can be done in $O(n^2)$ time.

Given two strings β and γ, let $lcp(\beta, \gamma)$ be the length of the longest prefix common to β and γ. Again as in [21], consider all the possible triples (L, M, R) that can arise in a binary search on the interval $[0, n^2 - 1]$ (here L, M, and R denote the left point, middle point, and right point of the interval that remains to be searched). There are exactly $n^2 - 2$ such triples, each with a unique midpoint $M \in [1, n^2 - 2]$ and for each triple $0 \leq L < M < R \leq n^2 - 1$. Let (L_M, M, R_M) be the unique triple containing midpoint M. Let $Llcp$ be an array of size $n^2 - 2$ such that $Llcp[M] = lcp(\alpha_{POS[L_M]}, \alpha_{POS[M]})$. Moreover, let $Rlcp$ be an array of size $n^2 - 2$ such that $Rlcp[M] = lcp(\alpha_{POS[R_M]}, \alpha_{POS[M]})$. These two arrays can be computed in $O(n^2)$ time as follows. We augment RLT_D with LCA data structures [17], [24] in $O(n^2)$ time. Now each entry of $Llcp$ and $Rlcp$ can be computed in constant time by an LCA query, for a total of $O(n^2)$ time. We have the following theorem.

THEOREM 10. *The tables POS, $Llcp$, and $Rlcp$ can be built in $O(n^2(\log|\Sigma| + \log n))$ time.*

Given a matrix PAT, we use the same search procedure of Manber and Myers [21] to find the subset of the $\alpha_{l,d}$, $0 \leq |d| < n$ and $1 \leq l \leq n - |d|$, that has pat as a prefix. As already pointed out, that is equivalent to finding all suffixes of $TEXT_d$, $0 \leq |d| < n$, that have pat as prefix. As in [21], such a procedure needs to compare characters of the $\alpha_{l,d}$ with characters of pat. pat can be explicitly built from PAT in $O(m^2)$ time. However, we do not construct the $\alpha_{l,d}$ explicitly. (It would be too costly.) Instead, we recover, when needed and in constant time, each character $\alpha_{l,d}[i]$ through the correspondence between $\alpha_{l,d}$, $\mathrm{L}suf_l(Lt_d)$, and the entries of $TEXT$. Thus, the time complexity of the procedure of Manber and Myers is unchanged and will depend on the size n^2 of our tables and the length m^2 of pat.

THEOREM 11. *Given that the arrays POS, $Llcp$, and $Rlcp$ have been computed for the matrix $TEXT$, finding all occurrences of PAT in $TEXT$ takes $O(m^2 + \log n + occ)$ time, where occ is the number of such occurrences.*

10. Application two: Dictionary matching. We are given a dictionary of s distinct square matrices $PAT_1, \ldots, PAT_s$, which we call patterns, where PAT_i is of dimension $m_i \times m_i$, $1 \leq i \leq s$. We are allowed to preprocess the dictionary (preprocessing step). Then we are given a square matrix $TEXT$ of dimension $n \times n$, which we call text. We must report all occurrences of patterns of the dictionary in the text (search step). The search step may be repeated with different texts.

10.1. Preprocessing step. Our preprocessing step consists of building a compacted trie LT over the alphabet $\mathrm{L}\Sigma$ that represents, as Lstrings, all suffixes of $PAT_1\hat{\$}_1, \ldots, PAT_s\hat{\$}_s$. Let $\mathrm{L}pat_1\hat{\$}_1, \ldots, \mathrm{L}pat_s\hat{\$}_s$ be the Lstrings corresponding to those matrices. Assume that an edge of LT is labeled by chunk $\mathrm{L}suf_i(\mathrm{L}pat_j\hat{\$}_j)[p, q]$. We represent such chunk in constant

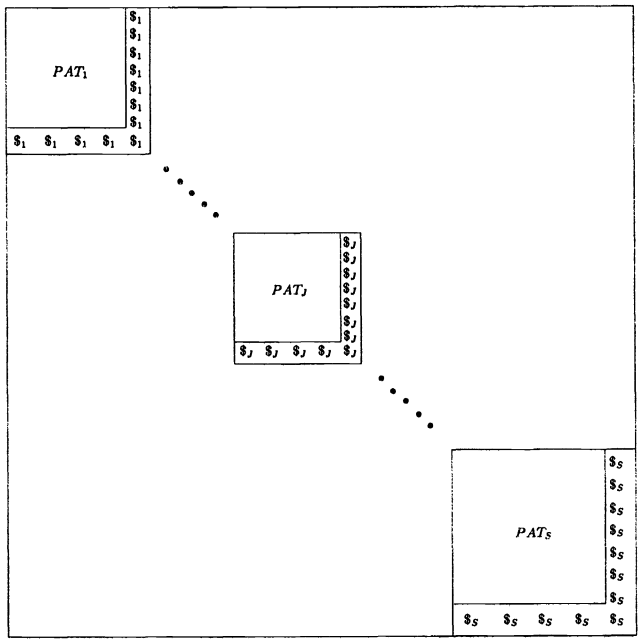


FIG. 16.

space with a quadruple (p, q, i, j). Using the quadruple, we can recover the starting points in $PAT_j\hat{\$}_j$ of each Lcharacter in the chunk. (The approach is the same as the one in §3.2.3, where we have shown how, from quadruples, we can recover starting points in A of each Lcharacter of chunks coming from the Lstrings corresponding to A_d, $0 \leq |d| < n$, and its suffixes.)

In order to build LT, we use the same approach of as in §7, where we have shown how to build the compacted trie LT_D representing, as Lstrings, all suffixes of matrices A_d, $1 \leq |d| < n$. Indeed, we think of $PAT_1\hat{\$}_1, \ldots, PAT_s\hat{\$}_s$ as being aligned, in that order, on the main diagonal of a matrix B (see Fig. 16). B is of dimension $\hat{n} = \Sigma_{i=1}^{s}(m_i + 1)$ and it has dummy symbols in its entries not covered by any $PAT_j\hat{\$}_j$. Note that we do not actually need to build B explicitly since we can recover its entries from the pattern matrices. In what follows, we omit the details and proofs since the analogy with the construction of §7 is obvious. (The role played here by PAT_j, $1 \leq j \leq s$, is the same as the role played by A_d, $0 \leq |d| < n$, in §7.)

Let LT_b be the Lsuffix tree for Lstring $Lb\hat{\$}$ and let $\hat{q}$ be the leaf of LT_b that is the locus of the Lstring corresponding to $B[\hat{n} + 1, \hat{n} + 1] = \$$, i.e., the leaf that is the locus of the Lstring corresponding to the last suffix of $B\hat{\$}$. Using exactly the same arguments as in Lemma 11, we can show that LT is isomorphic to $\mathrm{L}T_b - \{\hat{q}\}$. Thus we can obtain LT by building $\mathrm{L}T_b - \{\hat{q}\}$ and then transforming the triples on its edges into quadruples representing chunks of $\mathrm{L}pat_j$, $1 \leq j \leq s$, or of its Lsuffixes. This latter step is similar to the transformation of $\mathrm{L}T_b - \{q\}$ into LT_D described in §7 and we omit the details. We point out that in transforming $\mathrm{L}T_b$ into LT, we keep all the auxiliary data structures that are used in §§5 and 6 to build $\mathrm{L}T_b$. In particular, for each leaf $q \in LT$, $SL(q)$ is a suffix link defined exactly as for $\mathrm{L}T_b$ and its value is known at the end of the computation (because $\mathrm{L}T_b$ satisfies Invariant 1). Moreover, LT is a dynamic tree and each node of LT points to some arbitrary leaf in its subtree. Note also that the size of LT is $O(\hat{n})$, since LT_b is the Lsuffix tree of an Lstring of length $O(\hat{n})$ and, by Theorem 1, LT_b has size $O(\hat{n})$.

As for the efficient construction of LT_b, recall that we need to be able to compare, in constant time, two Lcharacters of equal length as strings of Σ^* and each coming from $Lb\hat{\$}$ or one of its Lsuffixes. Note that the chunks (and in particular the Lcharacters) of $Lb\hat{\$}$ and of its Lsuffixes have a relevant part and a dummy part, which are defined as in §7 except that the matrices A_d's are replaced by the matrices PAT_j's. Again as in §7, we consider only the relevant parts of Lcharacters coming from Lb or one of its Lsuffixes. Thus the problem of comparing two Lcharacters of equal length as strings of Σ^* with each coming from $Lb\hat{\$}$ or one of its Lsuffixes boils down to the efficient comparison of two Lcharacters of equal length as strings of Σ^* and each coming from $Lpat_j$, $1 \leq j \leq s$, or one of its Lsuffixes. In order to compare two such Lcharacters in constant time, we proceed as in §4. However, the suffix tree T_{rows} (T_{cols}, respectively) is the suffix tree of the string obtained by concatenating (separated by \$'s) all rows (columns, respectively) of matrices PAT_j, $1 \leq j \leq s$. Notice that there is a unique leaf such that the concatenation of the labels on the path from $root(T_{rows})$ ($root(T_{cols})$, respectively) to that leaf gives $PAT_{j_1}[i_1, g_1 : m_{j_1}]$ ($PAT_{j_1}[i_1 : m_{j_1}, g_1]$, respectively). Such construction takes $O(\hat{n} \log |\Sigma|)$ time because $rows$ and $cols$ are strings of length $O(\hat{n})$ [22]. We augment both T_{rows} and T_{cols} with LCA data structures [17], [24]. As in §4, the comparison of two Lcharacters, each coming from $Lpat_j$, $1 \leq j \leq s$, or one of its Lsuffixes, reduces to a constant number of LCA queries and therefore takes constant time. Thus the comparison of two Lcharacters, each coming from $Lb\hat{\$}$ or one of its Lsuffixes, also takes constant time. Now, by Theorem 4, we can build LT_b in $O(\hat{n} \log \hat{n})$ time.

We need to augment LT with additional data structures that will be useful during the search step. We mark each node of LT that is a locus of some Lstring corresponding to some PAT_g, $1 \leq g \leq s$. For each node $w \in LT$, we set a *prefix link* to its lowest marked proper ancestor. Such data structures can be computed in $O(\hat{n})$ time by a bottom up visit of LT. We finally note that since LT is a compacted trie for the Lsuffixes of $Lpat_j\hat{\$}_j$, $Lpat_j$ may not have a locus defined in LT. That happens only when PAT_j is not a prefix of any other PAT_i because $\hat{\$}_j$ does not match $\hat{\$}_i$, $i \neq j$. In that case $Lpat_j$ has an extended locus, which corresponds to the leaf u, locus of $Lpat_j\hat{\$}_j$. We assume that, in such a case, the length of $Lpat_j$ is stored at u and that u is marked. The computation of this information can be easily done during the computation of LT, leaving the time bounds unchanged. The above discussion outlines the proof of the following theorem.

THEOREM 12. *Given a set of matrices $PAT_1, \ldots, PAT_s$, each of dimension $m_i \times m_i$, $1 \leq i \leq s$, the preprocessing step for the dictionary matching problem takes $O(\hat{n}(\log \hat{n} + \log |\Sigma|))$ time, where $\hat{n} = \Sigma_{i=1}^{s}(m_i + 1)$.*

10.2. Search step. Given an $n \times n$ matrix $TEXT$, we want to use LT to find all occurrences of the patterns of the dictionary in the text. Let $TEXT_d$ and Lt_d, $0 \leq |d| < n$, be defined as in §9.1.

For a fixed d, $1 \leq i \leq n - |d|$, let $k = \max(d + 1, 1)$, $l = \max(1, -d + 1)$, $k' = \min(n, n + d)$, and $l' = \min(n, n - d)$. For $i = 1 \ldots n - |d|$, let $Lh_{i,d}$ be the longest Lprefix of $Lsuf_i(Lt_d)$ that has an extended locus w in LT. That is, $Lh_{i,d}$ corresponds to the longest prefix of $TEXT[k + i - 1 : k', l + i - 1 : l']$ that is also a prefix of some matrix represented in LT as Lstring. Such matrices are all suffixes of patterns in the dictionary. In what follows, let $l_{i,d}$ be the length of $Lh_{i,d}$.

We first show how to find all occurrences of patterns of the dictionary in position $(k + i - 1, l + i - 1)$ of the text, given $Lh_{i,d}$. Then we address the problem of computing $Lh_{i,d}$ for a fixed d and for $i = 1 \ldots n - |d|$. Finally, we put all the pieces together.

10.3. Finding occurrences in one position of *TEXT*. Fix i and d, $0 \leq |d| < n$ and $1 \leq i \leq n - |d|$, and let k, k', l, and l' be defined as above. We show how to compute all

occurrences of patterns of the dictionary in position $(k+i-1, l+i-1)$ of $TEXT$, assuming that we know $Lh_{i,d}$ and its extended locus w in LT. The following procedure takes in input i, d, the length of $Lh_{i,d}$ and w. Its correctness and time analysis are addressed in Lemma 15. (Since its proof is analogous to the one-dimensional case reported in [5], we omit it.)

Procedure *occurrence*

1 If w is a marked internal node and it is the locus of $Lh_{i,d}$, or it is a marked leaf and the length stored at that leaf is equal to the length of $Lh_{i,d}$, then report an occurrence at (k, l).
2 Starting from w and following the prefix links, visit all marked internal nodes on the path from $root(LT)$ to w. For each of them, report an occurrence at (k, l).

LEMMA 15. *Given i, d, the length of $Lh_{i,d}$, and w, procedure occurrence correctly finds in $O(occ_{i,d}+1)$ time, the $occ_{i,d}$ occurrences of patterns of the dictionary that start at position $(k+i-1, l+i-1)$ of $TEXT$.*

10.4. Computing $Lh_{i,d}$, d fixed and $i = 1 \ldots n-|d|$. The approach that we use in computing $Lh_{i,d}$, for $i = 1 \ldots n-|d|$, is essentially the same that we used in §§5 and 6 to compute $Lhead_i$. Recalling the notation and definitions used in §5, the analogy is the following. LT_{i-1} can be seen as a compacted trie representing (as Lstrings) the first $i-1$ suffixes of $A[1 : n+1, 1 : n+1]$. Thus $Lhead_i$ corresponds to the longest prefix of $A[i : n+1, i : n+1]$ that is also prefix of some matrix represented by LT_{i-1} as Lstrings. LT can be seen as a compacted trie representing (as Lstrings) all suffixes of patterns in the dictionary. But $Lh_{i,d}$ is the longest Lprefix of $Lsuf_i(Lt_d)$ that has an extended locus in LT. Thus $Lh_{i,d}$ corresponds to the longest prefix of $TEXT[k+i-1 : k', l+i-1 : l']$ that is also prefix of some matrix represented in LT as Lstring.

Recall that the computation of $Lhead_i$ is done in order of increasing i. Moreover, it is based on the relationship between $Lhead_i$, $Lhead_{i-1}$, and the information stored in LT_{i-1}. Indeed, starting from the contracted locus of $Lhead_{i-1}$ in LT_{i-1}, we first find the contracted locus of $L\alpha$, which is the second suffix of $Lhead_{i-1}$ and a prefix of $Lhead_i$. We do rescanning only when the contracted locus of $Lhead_{i-1}$ is not $root(LT_{i-1})$. Then, starting from the contracted locus of $L\alpha$, we find the rest of $Lhead_i$. (This is scanning.)

Analogous to that, the computation of $Lh_{i,d}$ is also done in order of increasing i, $i = 1 \ldots n-|d|$. Moreover, it is based on the relationship between $Lh_{i,d}$, $Lh_{i-1,d}$, and the information stored in LT. Indeed, let $h = \min(0, l_{i-1,d}-1)$ and let $L\gamma$ be the second Lsuffix of $Lh_{i-1,d}$. Since $Lh_{i-1,d}$ corresponds to $TEXT[k+i-2 : k+i+l_{i-1,d}-3, l+i-2 : l+i+l_{i-1,d}-3]$, $L\gamma$ corresponds to $TEXT[k+i-1 : k+i+h-2, l+i-1 : l+i+h-2]$ and it is the analog of $L\alpha$ above. Starting from the extended locus of $Lh_{i-1,d}$ in LT, we first find the contracted locus of $L\gamma$. Here there is also a rescanning phase (similar to the one of §§5 and 6) which is done only when the extended locus of $Lh_{i-1,d}$ is not $root(LT)$, i.e., when both $Lh_{i-1,d}$ and $L\gamma$ are not empty. Then, from the contracted locus of $L\gamma$, we find the rest of $Lh_{i,d}$. This is the scanning phase and it will be much simpler than the one in §§5 and 6 since there is no need to update either LT or any auxiliary data structure.

The procedure that we use to find $Lh_{i,d}$ is a stripped down version of **STI**. It takes in input v, the extended locus of $Lh_{i-1,d}$. For $i = 1$, it takes in input $root(LT)$, which is the extended locus of the empty Lstring $Lh_{0,d}$. Scanning and rescanning are implemented as in **STI**. Therefore, we omit here a detailed discussion of such implementations.

Procedure DSTI—Skeleton iteration *i*

1. If $v = root(LT)$ then $u := root(LT)$ and skip *rescanning*.
2. *Rescanning:* find the contracted locus u of $L\gamma$, given the extended locus $v \neq root(LT)$ of $Lh_{i-1,d}$ and SL for all leaves in LT.
3. *Scanning:* starting from u and skipping $L\gamma$, find the extended locus of $Lh_{i,d}$.

LEMMA 16. *Fix d, $1 \leq |d| < n$. Given the Lstring Lt_d, corresponding to matrix $TEXT[k : k', l : l']$, and LT, the compacted trie over the alphabet $L\Sigma$ representing (as Lstrings) all suffixes of matrices in the dictionary,* **DSTI** *correctly computes $Lh_{i,d}$, for $i = 1 \ldots n - |d|$. It takes $O((n - |d|) \log \hat{n})$ time.*

Proof. The proof of correctness is by induction. It is similar to the proof of correctness reported in Theorem 4 and it is omitted. As for the time analysis, using the techniques of §6, the cost of the ith iteration is bounded by $O((l_i - \min(0, l_{i-1} - 1) + 1) \log \hat{n})$ time. Adding over all iterations, we get the claimed bounds because $l_{1,d} \leq n - |d|$ and $l_{n-|d|,d} \leq 1$. □

10.5. Putting the pieces together. In this subsection we first address the issue on how to compare, in constant time, Lcharacters coming from Lstrings representing suffixes of patterns in the dictionary with Lcharacters coming from Lstrings representing suffixes of $TEXT_d$, $0 \leq |d| < n$. Then we derive the time complexity of the whole search procedure.

Again, we use quadruples to represent chunks coming from Lt_d, $0 \leq |d| < n$. The quadruple (p, q, i, d) represents the chunk $Lsuf_i(Lt_d)[p, q]$. We can recover, in constant time, the starting points in $TEXT$ of an occurrence of any of the Lcharacters from such a chunk. We use the same approach as in §3.2.3, where we have shown how, given a quadruple, we can recover in constant time the starting points in A of an occurrence of any of the Lcharacters from the chunk represented by the quadruple.

In order to achieve the desired time bound to compare two Lcharacters, one coming from Lstrings representing patterns and the other coming from Lstrings representing submatrices of $TEXT$, we need to express the rows (columns, respectively) of $TEXT$ in function of the rows (columns, respectively) of the patterns. Indeed, consider $TEXT[g, f : n]$ ($TEXT[g : n, f]$, respectively). Among the rows (columns, respectively) of the pattern matrices, we compute the subrow (subcolumn, respectively) that has a prefix of maximum length in common with $TEXT[g, f : n]$ ($TEXT[g : n, f]$, respectively). In formulae, we compute a quadruple $(e, \hat{v}, r, j)$ such that $PAT_j[e, \hat{v} : \hat{v} + r - 1]$ ($PAT_j[e : e + r - 1, \hat{v}]$, respectively) is the longest subrow (subcolumn, respectively) of patterns that is a prefix of $TEXT[g, f : n]$ ($TEXT[g : n, f]$, respectively). For each entry (g, f) of $TEXT$, we store such quadruple in an array $WHERER$ ($WHEREC$, respectively). Given that during the preprocessing step we have computed T_{rows} (T_{cols}, respectively), the computation of $WHERER$ ($WHEREC$, respectively) is a standard string matching task [11] and it can be done in $O(n^2 \log |\Sigma|)$ time.

Let us assume that we want to compare the Lcharacter $Lsuf_{i_1}(Lpat_{j_1})[q] = \delta$ with the Lcharacter $Lsuf_i(Lt_d)[q] = \epsilon$. Note that $\delta = \delta_1\delta_2$, where $\delta_1 = PAT_{j_1}[i_1+q-1, i_1 : i_1+q-2]$ and $\delta_2 = PAT_{j_1}[i_1 : i_1 + q - 1, i_1 + q - 1]$. Moreover, $\epsilon = \epsilon_1\epsilon_2$, where $\epsilon_1 = TEXT[k+i+q-2, l+i-1 : l+i+q-3]$ and $\epsilon_2 = TEXT[k+i-1 : k+i+q-2, l+i+q-2]$. Using the table $WHERER$ ($WHEREC$, respectively), we know where the longest prefix ω_1 (ω_2, respectively) of $TEXT[k + i + q - 2 : l + i - 1 : n]$ ($TEXT[k + i - 1 : n, l + i + q - 2]$, respectively) occurs as a subrow (subcolumn, respectively) of some pattern matrix. Say it occurs in position (c_1, c_2) of PAT_r. Using an LCA query involving the leaves of T_{rows} (T_{cols}, respectively) associated with $PAT_r[c_1, c_2 : m_r]$ ($PAT_r[c_1 : m_r, c_2]$, respectively) and $PAT_{j_1}[i_1 + q - 1, i_1 : m_{j_1}]$ ($PAT_{j_1}[i_1 : m_{i_1}, i_1 + q - 1]$, respectively), we can compute in constant time the longest prefix common to ω_1 (ω_2, respectively) and δ_1 (δ_2, respectively). Using such information, we can compute in constant time whether δ_1 (δ_2, respectively) is equal to ϵ_1 (ϵ_2, respectively), therefore establishing, in constant time, whether the two Lcharacters are equal.

In order to find all occurrences of patterns of the dictionary in the text, we first compute the arrays $WHERER$ and $WHEREC$ defined above. This will cost a total of $O(n^2 \log |\Sigma|)$ time. Then we use the algorithm of §10.4 to compute $Lh_{i,d}$, for $i = 1 \ldots n - |d|$ and $0 \leq |d| < n$. Summing the bound of Lemma 16 over all d's, this will cost a total of $O(n^2 \log \hat{n})$ time.

For each $Lh_{i,d}$, we report all occurrences in the corresponding position of $TEXT$ using the procedure of §10.3. Summing the bound of Lemma 15, for $i = 1 \cdots n - |d|$ and $0 \leq |d| < n$, this will cost a total of $O(occ + n^2)$ time, where occ is the number of occurrences of patterns in the text. We obtain the following theorem.

THEOREM 13. *Given LT, the compacted trie representing (as Lstrings) all suffixes of matrices in the dictionary, it takes $O(n^2 \log \hat{n} + occ)$ time to find the occ occurrences of pattern matrices of the dictionary in an $n \times n$ matrix $TEXT$.*

11. Concluding remarks and open problems. We have shown how to efficiently build the Lsuffix tree and its refinement for an $n \times n$ matrix A. We have then discussed applications of this new data structure to the pattern retrieval problem and to dictionary matching. It would be interesting to obtain algorithms for the construction of the Lsuffix tree and of the RLsuffix tree with time complexity linear in the input size. That would provide solutions to the pattern retrieval and dictionary matching problems with time complexities analogous to the ones known for the one-dimensional problems.

Acknowledgments. I would like to thank Brenda Baker for many helpful discussions and comments related to this paper.

REFERENCES

[1] A. AHO, *Algorithms for finding pattern in strings*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., Elsevier Science Publishers B.V., New York, 1990, pp. 257–295.

[2] A. AHO AND M. CORASICK, *Efficient string matching: An aid to bibliographic search*, Comm. ACM, 18 (1975), pp. 333–340.

[3] A. AMIR, G. BENSON, AND M. FARACH, *Alphabet independent two dimensional matching*, in Proc. 24th Symposium on Theory of Computing, ACM, 1992, pp. 59–68.

[4] A. AMIR AND M. FARACH, *Two-dimensional dictionary matching*, Inform. Process. Lett., 44 (1992), pp. 233–239.

[5] A. AMIR, M. FARACH, Z. GALIL, R. GIANCARLO, AND K. PARK, *Fully dynamic dictionary matching*, J. Comput. System Sci., 49 (1994), pp. 208–222.

[6] A. AMIR, M. FARACH, R. IDURY, J. L. POUTRE, AND A. SCHAFFER, *Improved dynamic dictionary matching*, in Proc. Fourth Symposium on Discrete Algorithms, ACM-SIAM, 1993, pp. 392–401.

[7] B. S. BAKER, *A theory of parameterized pattern matching: Algorithms and applications*, in Proc. 25th Symposium on Theory of Computing, ACM, 1993, pp. 71–80.

[8] T. BAKER, *A technique for extending rapid exact string matching to arrays of more than one dimension*, SIAM J. Comput., 7 (1978), pp. 533–541.

[9] R. BIRD, *Two dimensional pattern matching*, Inform. Process. Lett., 6 (1978), pp. 168–170.

[10] M. T. CHEN AND J. SEIFERAS, *Efficient and elegant subword-tree construction*, in Combinatorial Algorithms on Words, NATO ASI Series F, Vol. 12, A. Apostolico and Z. Galil, eds., Springer-Verlag, Berlin, 1984, pp. 97–107.

[11] Z. GALIL AND R. GIANCARLO, *Data structures and algorithms for approximate string matching*, J. Complexity, 4 (1988), pp. 32–72.

[12] Z. GALIL AND K. PARK, *A truly alphabet independent two-dimensional pattern matching algorithm*, in Proc. 33th Symposium on Foundations of Computer Science, IEEE, 1992, pp. 247–256.

[13] R. GIANCARLO, *Dynamic maintenance of the Lsuffix tree, with applications*, in preparation.

[14] ———, *An index data structure for matrices, with applications to fast two-dimensional pattern matching*, in Lecture Notes in Computer Science, Vol. 709, Springer-Verlag, Berlin, 1993, pp. 337–348.

[15] ———, *The suffix tree of a square matrix, with applications*, in Proc. Fourth Symposium on Discrete Algorithms, ACM-SIAM, 1993, pp. 402–411.

[16] G. GONNET, *Efficient searching of text and pictures- extended abstract*, Tech. Report, University of Waterloo, OED-88-02, 1988.

[17] D. HAREL AND R. TARJAN, *Fast algorithms for finding nearest common ancestors*, SIAM J. Comput., 13 (1984), pp. 338–355.

[18] I. IDURY AND A. SCHAFFER, *Multiple matching of rectangular patterns*, in Proc. 25th Symposium on Theory of Computing, ACM, 1993, pp. 81–90.

[19] R. JAIN, *Workshop report on visual information systems*, Tech. Report, National Science Foundation, 1992.

[20] D. KNUTH, *The Art of Computer Programming, Vol.* 3: *Sorting and Searching*, Addison-Wesley, Reading, MA. 1973.
[21] U. MANBER AND E. MYERS, *Suffix arrays: A new method for on-line string searches*, in Proc. First Symposium on Discrete Algorithms, ACM-SIAM, 1990, pp. 319–327.
[22] E. MCCREIGHT, *A space economical suffix tree construction algorithm*, J. Assoc. Comput. Mach., 23 (1976), pp. 262–272.
[23] A. ROSENFELD AND A. KAK, *Digital Picture Processing*, Academic Press, New York, 1982.
[24] B. SCHIEBER AND U. VISHKIN, *On finding lowest common ancestors: Simplification and parallelization*, SIAM J. Comput., 17 (1988), pp. 1253–1262.
[25] D. SLEATOR AND R. TARJAN, *A data structure for dynamic trees*, J. Comput. System Sci., 26 (1983), pp. 362–391.
[26] P. WEINER, *Linear pattern matching algorithms*, in Proc. 14th Symposium on Switching and Automata Theory, IEEE, 1973, pp. 1–11.

SIAM J. COMPUT.
Vol. 24, No. 3, pp. 563–578, June 1995

010

EFFICIENT ALGORITHMS FOR THE HITCHCOCK TRANSPORTATION PROBLEM*

TAKESHI TOKUYAMA† AND JUN NAKANO†

Abstract. We consider the Hitchcock transportation problem on n supply points and k demand points when n is much greater than k. The problem can be solved in $O(kn^2 \log n + n^2 \log^2 n)$ time if an efficient minimum-cost flow algorithm is directly applied. Applying a geometric method named *splitter finding* and a randomization technique, we can improve the time complexity when the ratio c of the maximum supply to the minimum supply is sufficiently small. The expected running time of our randomized algorithm is $O(\frac{kn \log cn}{\log(n/k^4 \log^2 k)})$ if $n > k^4 \log^2 k$, and $O(k^5 \log^2 n \log cn)$ if $n \leq k^4 \log^2 k$. If $n = \Omega(k^{4+\epsilon})$ $(\epsilon > 0)$ and $c = \text{poly}(n)$, the problem is solved in $O(kn)$ time, which is optimal.

Key words. transportation problem, computational geometry, randomized algorithm

AMS subject classifications. 68U05, 90B06, 90B80, 90C08

1. Introduction.

Consider a complete directed bipartite graph $\mathcal{G} = K(n, k)$ with n source nodes $s_1, \ldots, s_n$ and k demand nodes $t_1, \ldots, t_k$. The node s_i has a supply of size ω_i for $i = 1, 2, \ldots, n$, and t_j has a demand of size λ_j for $j = 1, 2, \ldots, k$. Each directed edge (s_i, t_j) has a cost $\alpha_{i,j}$, which is charged for each unit of flow on it. The problem of finding the minimum cost flow from supply points to demand points is called the Hitchcock transportation problem [4]–[6] (Fig. 1.1).

For an intuitive example, imagine a distributed data system with k storage devices D_1, D_2, ..., D_k and n data $z_1, z_2, \ldots, z_n$. The sizes of D_j and z_i are λ_j and ω_i, respectively. Each data is called by processors, and if data z_i is placed in D_j, the (expected) communication time for calling a unit of data z_i is known to be $\alpha_{i,j}$. The problem of finding the allocation of data that minimizes the communication cost is formulated into the Hitchcock transportation problem. The Hitchcock transportation problem can be formulated as an instance of linear programming. Its standard form is as follows:

$$\text{Minimize} \quad \sum_{i=1}^{n} \sum_{j=1}^{k} \alpha_{i,j} y_{i,j} \tag{1.1}$$

subject to

$$\sum_{i=1}^{n} y_{i,j} = \lambda_j \quad (j = 1, 2, \ldots, k), \tag{1.2}$$

$$\sum_{j=1}^{k} y_{i,j} = \omega_i \quad (i = 1, 2, \ldots, n), \tag{1.3}$$

$$y_{i,j} \geq 0 \quad (i = 1, 2, \ldots, n;\ j = 1, 2, \ldots, k). \tag{1.4}$$

Because of the symmetry of the problem, we can assume that $n \geq k$ without loss of generality. The problem is feasible if and only if $\sum_{j=1}^{k} \lambda_j = \sum_{i=1}^{n} \omega_i$. However, we can relax this feasibility condition to $\sum_{j=1}^{k} \lambda_j \leq \sum_{i=1}^{n} \omega_i$ and replace (1.2) with the inequalities

$$\sum_{j=1}^{k} y_{i,j} \leq \omega_i \quad (i = 1, 2, \ldots, n),$$

*Received by the editors August 31, 1992; accepted for publication (in revised form) December 31, 1994.

†IBM Research, Tokyo Research Laboratory, 1623-14, Shimo-tsuruma, Yamato-shi, Kanagawa 242, Japan (ttoku@vnet.ibm.com and nakanoj@vnet.ibm.com).

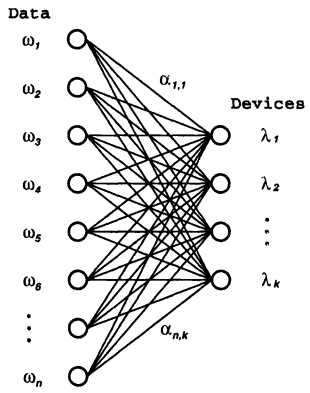


FIG. 1.1. *The Hitchcock transportation problem.*

since the relaxed problem is transformed into the standard form by defining a virtual demand point t_{k+1} such that $\lambda_{k+1} = \sum_{i=1}^{n} \omega_i - \sum_{j=1}^{k} \lambda_j$ and $\alpha_{i,k+1} = 0$ for $i = 1, 2, \ldots, n$.

The Hitchcock transportation problem is a kind of capacitated minimum-cost flow problem, which is known to be soluble in strongly polynomial time [10], [11], [16], [18]. Indeed, the problem is the minimum-cost flow problem on a network with $N = n + k$ nodes and $M = kn$ edges. The best existing algorithm [16] for solving the uncapacitated minimum-cost flow problem is $O(N \log N(M + N \log N))$, which solves the Hitchcock transportation problem in $O(kn^2 \log n + n^2 \log^2 n)$ time if $n \geq k$. This complexity is more than the square of n even if k is very small.

In many applications, the number of supply points (n) is often much greater than that of demand points (k), or vice versa. Indeed, in the example shown above, the number of data is usually much greater than that of devices. Matsui [13] gives a linear-time algorithm for solving the Hitchcock transportation problem with respect to n if k can be considered as a constant. However, the time complexity of his algorithm is $O(n(k!)^2)$ if k is not a constant; thus it is too expensive unless k is extremely small. Recently, Ahuja et al. [1] studied network flow problems in unbalanced bipartite graphs (where $k \ll n$), and obtained an $O(k^2 n \log(nC))$ time algorithm for the Hitchcock transportation problem (where $C = \max\{|\alpha_{i,j}|\}$ is the maximum edge cost of the associated bipartite graph).

If $\omega_i = 1$ for all $i = 1, 2, \ldots, n$, the problem is named the λ-assignment problem. Tokuyama and Nakano [20] gave a geometric approach called *splitter finding* to the λ-assignment problem and solved it in $O(nk + n^{0.5}k^{3.5})$ expected time, which they have recently improved to $O(nk + k^{2.5}n^{0.5} \log^{1.5} n)$ expected time [19].

In this paper, we first give a geometric interpretation of the problem and show that Orlin's algorithm can solve the Hitchcock transportation problem in $O(k^2 n \log^2 n)$ time if it is efficiently implemented. The main result is that if there is a constant c such that $1 \leq \omega_i \leq c$ for $1 \leq i \leq n$, we can design an efficient randomized algorithm with a new pruning technique, whose running time is

$$\begin{cases} O\left(\dfrac{kn \log cn}{\log(n/k^4 \log^2 k)}\right), & \text{when } n > k^4 \log^2 k, \\ O(k^5 \log^2 n \log cn). & \text{when } n \leq k^4 \log^2 k, \end{cases}$$

with a probability larger than $1 - n^{-\gamma}$ for any positive constant γ. If $n = \Omega(k^{4+\epsilon})$ ($\epsilon > 0$) and c is polynomial in n, this algorithm runs in $O(kn)$ time, which is optimal.

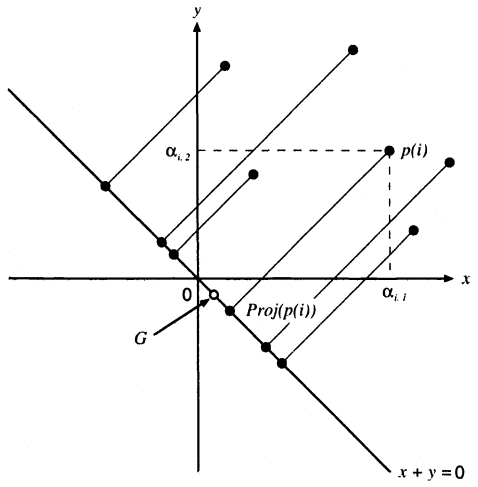


FIG. 2.1. *Geometric transformation in two-dimensional space.*

2. Geometric interpretation. Given a Hitchcock transportation problem defined by (1.1), ..., (1.4), we consider the following geometric model. For each index i ($i = 1, 2, \ldots, n$), we define a point $p(i) = (\alpha_{i,1}, \ldots, \alpha_{i,k})$ in k-dimensional real space $\mathbf{R}^k$. Let $S = \{p(1), p(2), \ldots, p(n)\}$. We call ω_i the *weight* of $p(i)$ (it is often called the *supply* of $p(i)$ in the literature).

In the bipartite graph $\mathcal{G}$, if an amount of ξ out of the weight of the supply node s_i flows on the edge (i, j), we say that a ξ portion of the point $p(i)$ is *assigned* to the jth demand point t_j.

Before dealing with general cases, we first consider the very easy case where $k = 2$ and give an intuition of our idea. If $k = 2$, S is a set of n points on a plane. If a ξ portion of $p(i)$ is assigned to t_1 and $\omega_i - \xi$ portion is assigned to t_2, the cost for the point $p(i)$, which is the sum of the costs on the edges $(i, 1)$ and $(i, 2)$ on $\mathcal{G}$, is $\xi\alpha_{i,1} + (\omega_i - \xi)\alpha_{i,2}$. Therefore, if we increase ξ by a unit size, the cost increases by $\alpha_{i,1} - \alpha_{i,2}$, which we call the *reduced cost* of the edge $(i, 1)$. Hence, the problem is solved by the following way: first we compute the reduced costs of the edge $(i, 1)$ for $i = 1, 2, \ldots, n$. Then, we assign the points to t_1 in smallest-first fashion with respect to the reduced cost until the weight sum becomes λ_1. Then, we assign the other points to t_2. Notice that there is at most one supply node that has positive flows to both t_1 and t_2.

Geometrically, if we project the point $p(i)$ orthogonally on the line $x + y = 0$, we get a point $\text{Proj}(p(i)) = ((\alpha_{i,1} - \alpha_{i,2})/2, (\alpha_{i,2} - \alpha_{i,1})/2)$ (Fig. 2.1). The reduced cost is proportional to the x-coordinate value of $\text{Proj}(p(i))$. We choose a point G on the line $x + y = 0$ such that satisfying the total weight of the points on the left of G is at most λ_1, and that on the right of it is at most λ_2. We call this point the λ-splitter. Hence, the Hitchcock transportation problem is equivalent to the problem of computing the splitter if $k = 2$, which can be solved by the randomized selection algorithm [17].

We generalize the idea to the higher-dimensional case. We project each point $p(i) = (\alpha_{i,1}, \ldots, \alpha_{i,k})$ orthogonally onto the hyperplane L defined by the equation $x_1 + x_2 + \cdots + x_k = 0$ (Fig. 2.2) and let $G = (g_1, g_2, \ldots, g_k)$ be a point on L. For each j ($j = 1, \ldots, k$), a closed region satisfying $x_j - g_j \leq x_h - g_h$ for all $h \neq j$ is called the jth region split by G, which is denoted by $T(G; j)$. The space subdivision of $\mathbf{R}^k$ into $T(G; j)$ ($j = 1, 2, \ldots, k$) is called a *splitting* and G a *splitter* (Fig. 2.3). We project points of S onto L to obtain a point set that is

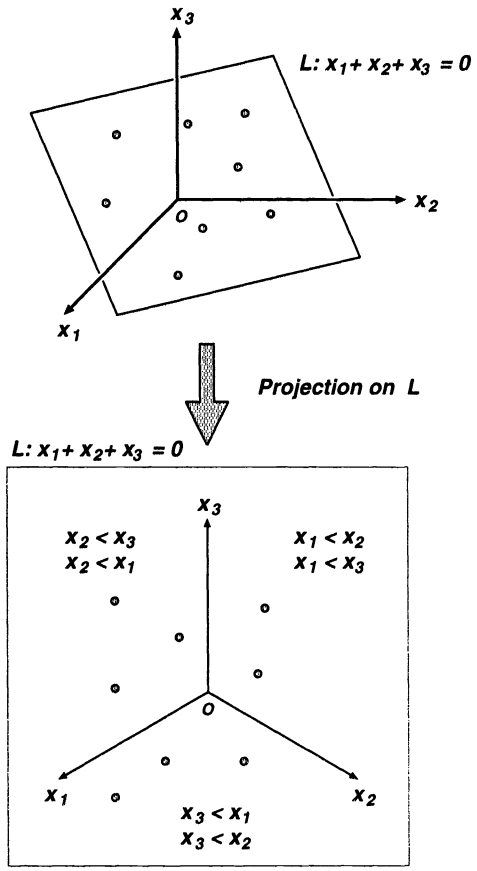


FIG. 2.2. *Geometric transformation in three-dimensional space.*

also denoted by S, but it is clear from the context, otherwise stated. The following theorem was originally proved by Numata and Tokuyama [15].

THEOREM 2.1 (existence of λ-splitter). *There exists a splitter G such that the sum of the weights of the points of S in $\bigcup_{j\in J} T(G; j)$ is greater than or equal to $\sum_{j\in J} \lambda_j$ for any $J \subset \{1, 2, \ldots, k\}$.*

Note that each $T(G; j)$ is a closed region containing its boundary. The splitter G defined in the theorem is called a *λ-splitter* of the point set S. The corresponding splitting is called a λ-splitting, which is a generalization of that given by the authors [20]. In the next section we show how to construct a λ-splitter incrementally.

THEOREM 2.2. *For a λ-splitter $G(g_1, \ldots, g_k)$, the Hitchcock transportation problem has a solution satisfying $y_{i,j} = \omega_i$ if $p(i)$ is in the interior of $T(G; j)$, $y_{i,j} = 0$ if $p(i)$ is outside $T(G; j)$, and $y_{i,j_1} + y_{i,j_2} = \omega_i$ if $p(i)$ is on the boundary between $T(G; j_1)$ and $T(G; j_2)$ (if $p(i)$ is contained in more than two closed regions $T(G; j_t)$ $(t = 1, \ldots, m)$, $\sum_{t=1}^{m} y_{i,j_t} = \omega_i$). Conversely, any solution of the Hitchcock transportation problem satisfies the above condition for a λ-splitter.*

Proof. It is easy to see that if we suitably divide the weights of boundary elements between regions, we can find a solution of (1.2), (1.3), and (1.4) that satisfies the conditions of the theorem. We will show that this solution minimizes the total cost. We define $\beta_{i,j} = \alpha_{i,j} - g_j$ for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, k$. Since $\sum_{i,j} \beta_{i,j} y_{i,j} = \sum_{i,j} \alpha_{i,j} y_{i,j} - \sum_j \lambda_j g_j$, the

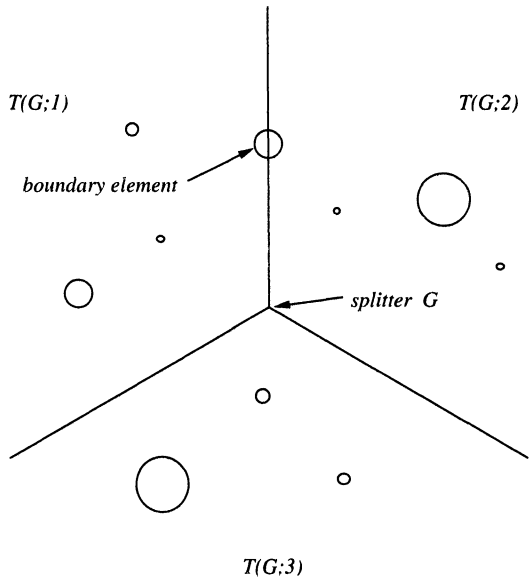


FIG. 2.3. *Splitter and splitting of point set projected onto the plane L ($k = 3$ and the area of each circle pictorially represents the weight of the corresponding point).*

optimal solution does not change if we replace $\alpha_{i,j}$ with $\beta_{i,j}$ for all i and j. In the new problem, the splitter coincides with the origin O. Since the region $T(O; j)$ is defined by $x_j \leq x_h$ for all $h \neq j$, it is clear that the above solution minimizes the total cost. The converse statement is obvious from the above argument. □

For the readers familiar with the linear programming, we note that the splitter can be defined by using the duality of LP and Theorem 2.2 is proved by using the *dual complementary slackness* condition [3], [12], [14]. Indeed, if we introduce variables u_i $(i = 1, 2, \ldots, n)$ and v_j $(j = 1, 2, \ldots, k)$ and define $\tilde{\omega}_{i,j} = \omega_{i,j} - u_i - v_j$, it is easy to see that the solution of the Hitchcock transportation problem is stable if we replace $\omega_{i,j}$ by $\tilde{\omega}_{i,j}$ for all i and j. The dual-complementary slackness condition says that we can choose u_i and v_j suitably so that

$$\tilde{\omega}_{i,j} = \min_{l=1,2,\ldots,k} \{\tilde{\omega}_{i,l}\}, \tag{2.1}$$

if the flow on the edge (i, j) of $\mathcal{G}$ is nonzero. (We omit the proof.) Since equation (2.1) is independent of u_i, we can set $u_i = (1/k) \sum_{j=1}^{k} \omega(i, j)$. Also, we can choose a solution of equation (2.1) satisfying $\sum_{j=1}^{k} v_j = 0$. Then, a solution $(v_1, \ldots, v_k)$ is a splitter, and vice versa. Note that the above fact has been used in the classical method for solving the Hitchcock transportation problem using *transportation tableau* [3], [14], where u_i is chosen as $\min_{j=1,\ldots,k}\{\omega_{i,j} - v_j\}$.

Theorem 2.2 implies that if we find a splitter, we can solve the Hitchcock transportation problem by finding a transportation of the points on the boundaries between regions. We call a λ-splitting a λ-*transportation* if we correctly assign the weights of the boundary elements.

Each region of a splitting has $k - 1$ boundary facets and there are $k(k - 1)/2$ facets in total. We say that S is simple if the number of incidence relations between the boundary facets and points is at most $k - 1$ for any splitter. Since the degree of freedom of G is $k - 1$, we can always make S simple by applying a small perturbation. For example, we may use the SOS system of Edelsbrunner [7]. Furthermore, the algorithm in this paper can be easily applied to

a nonsimple case almost directly without loss of time complexity. Thus, we assume from now on that S is simple.

Although the splitter is not unique, we may make the following observation.

PROPOSITION 2.3. *The solution of the Hitchcock transportation problem is unique if S is simple.*

3. Scaling algorithm. In this section, we give an efficient algorithm based on Orlin's minimum-cost flow algorithm [16]. Ours is an incremental algorithm that uses the scaling method. The main theorem in this section is as follows.

THEOREM 3.1. *The Hitchcock transportation problem is solvable in $O(k^2 n \log^2 n)$ time.*

In the rest of this section, which is devoted to proving the theorem, we assume that the weights are integers. Let the maximum weight be U and define $l = \lfloor \log_2 U \rfloor$. First, we design an $O(k^2 nl \log n)$ time algorithm, which resembles the Edmonds–Karp algorithm [8].

We decompose a point p into at most $l + 1$ points according to the 2-adic (binary) decomposition of its weight ω_p. The newly-created point with a weight 2^h is denoted by $p^{(h)}$ if its origin is the point p. Thus, we have a set $\tilde{S}$ of at most $n(l+1)$ points, each of which has a weight that is a power of 2. For a number a and an integer h, we define $a^{(h)} = 2^h \lfloor a/2^h \rfloor$, and extend this notation to vectors: $\lambda^{(h)} = (\lambda_1^{(h)}, \ldots, \lambda_k^{(h)})$. We thus obtain an incremental algorithm consisting of $l + 1$ stages.

```
HITCHCOCK
1.  G = origin;
2.  W = 0; {W is the total weight of the points inserted so far}
3.  for h = 0 to l;
    begin
3.1.    SPLIT(h, G, W);
    end;
end;
```

```
SPLIT(h, G, W)
1.  L = list of all the points of weight 2^(l-h);
2.  while L is not empty, and W < Σ_{j=1}^{k} λ_j^{(l-h)};
    begin
2.1.    choose a point p from L;
2.2.    add p in the region (say, T(G; s)) containing p in the current splitting;
2.3.    if the total weight assigned to T(G; s) exceeds λ_s^{(l-h)} then
2.3.1       UPDATE(G);
        end if;
2.4.    W = W + 2^(l-h);
    end while;
3.  split each remaining point in L into two points of weight 2^(l-h-1);
4.  return;
end;
```

The subroutine UPDATE(G) updates the current transportation so that no region overflows. Let the current splitter be $G = (g_1, g_2, \ldots, g_k)$. Let γ_j be the total weight currently assigned to $T(G; j)$; that is, G is a γ-splitter of the set of points added so far. Suppose that we are at the hth stage (i.e., executing SPLIT(h, G, W)); a new point is inserted into $T(G; s)$ and UPDATE is called. We say that an element of $T(G; j)$ is a *proper* element if some portion of its weight is assigned to $T(G; j)$ in the current transportation. Clearly, an interior element is a proper element.

We first assume the following two properties and show later that these properties hold throughout the algorithm.

PROPERTY 1. *There exists a region $T(G; t)$ such that $\gamma_t \leq \lambda_t^{(l-h)} - 2^{l-h}$.*

PROPERTY 2. *For each proper element of each region, the portion of its weight assigned to the region is an integer multiple of 2^{l-h}.*

For each ordered pair of two regions $T(G; a)$ and $T(G; b)$, we compute a point $q(a, b)$, which is the nearest proper element in $T(G; a)$ to the boundary between the two regions. Note that $q(a, b)$ may be a boundary element. We construct a complete directed graph $\mathcal{D}$ on the node set $\{v_1, v_2, \ldots, v_k\}$. For each directed edge (v_a, v_b), we give a (nonnegative) length $q(a, b)_b - q(a, b)_a - (g_b - g_a)$, where $q(a, b)_j$ is the jth coordinate value of $q(a, b)$.

Next, if the newly inserted point falls in the region $T(G; s)$, we compute the shortest path tree $\mathcal{T}$ of $\mathcal{D}$ rooted at v_s. Then, a new splitter $G' = (g'_1, \ldots, g'_k)$, which by definition satisfies $\sum_{j=1}^{k} g'_j = 0$, is defined by

$$g'_a - g'_b = q(a, b)_a - q(a, b)_b \tag{3.1}$$

for all pairs (a, b) such that (v_a, v_b) is a directed edge in the shortest path tree $\mathcal{T}$.

Since there are $k - 1$ adjacent relations in the tree, G' is uniquely determined. The following is a key lemma.

LEMMA 3.2. *A proper element of $T(G; j)$ is in $T(G'; j)$.*

Proof. Consider a point $x = (x_1, \ldots, x_k)$ located in the region $T(G; a)$. What we have to prove is that $x_a - x_b \leq q(a, b)_a - q(a, b)_b$ (for $b \neq a$) implies $x_a - x_b \leq g'_a - g'_b$, or $g'_a - g'_b \geq q(a, b)_a - q(a, b)_b$. If the edge (v_a, v_b) is in the shortest path tree $\mathcal{T}$, the claim is obvious from (3.1). If $(v_a, v_b) \notin \mathcal{T}$, we have $\text{dist}(s, b) \leq \text{dist}(s, a) + \text{length}(a, b)$, where $\text{dist}(s, a)$ denotes the distance from the root v_s to the node v_a in $\mathcal{T}$. Since for an edge $(v_i, v_j) \in \mathcal{T}$, $\text{length}(i, j) = q(i, j)_j - q(i, j)_i - (g_j - g_i) = g'_j - g'_i - (g_j - g_i)$, we obtain $\text{length}(a, b) \geq \text{dist}(s, b) - \text{dist}(s, a) = g'_b - g'_a - (g_b - g_a)$, that is, $g'_a - g'_b \geq q(a, b)_a - q(a, b)_b$. □

From this lemma, we are assured that we can find G' by translating the boundary facets without moving across any points in the current splitting as in Fig. 3.1.

COROLLARY 3.3. *G' is a γ-splitter of the point set before insertion.*

We will show that there is such a transportation after the insertion that the weight assigned to $T(G; t)$ is $\gamma_t + 2^{l-h}$ for some region satisfying Property 1 and that the weight assigned to $T(G; j)$ is γ_j for $j \neq t$.

We consider the shortest path from v_s to v_t, and push a flow of size 2^{l-h} along it. From the definition of G', if the flow is pushed from v_a to the adjacent node v_b, there exists a proper boundary element of $T(G'; a)$ on the boundary between $T(G'; a)$ and $T(G'; b)$. From Property 2, we can move 2^{l-h} of the weight of the boundary element from $T(G'; a)$ to $T(G'; b)$. Thus, we can push the weight successively according to the shortest path so that the weight of $T(G; s)$ is diminished by 2^{l-h} and that of $T(G; t)$ is increased by 2^{l-h}.

Hence, we can update the transportation so that no region overflows if Properties 1 and 2 hold throughout the algorithm.

CLAIM 3.4. *Property 1 and Property 2 hold throughout the algorithm.*

Proof. Obviously, these properties hold initially. If Property 2 holds in one stage of HITCHCOCK (i.e., for some h in step 3), then it clearly holds in the next stage. Thus, it suffices to show that update procedure UPDATE(G) preserves Property 2. Since the weight of a point assigned to a region is changed by 2^{l-h}, UPDATE(G) preserves Property 2.

Because of the weight check in step 2 of SPLIT(h, G, W), the total weight assigned so far is less than $\sum_{j=1}^{k} \lambda_j^{(l-h)}$, and thus less than or equal to $(\sum_{j=1}^{k} \lambda_j^{(l-h)}) - 2^{l-h}$. Thus, Property 1 follows Property 2. □

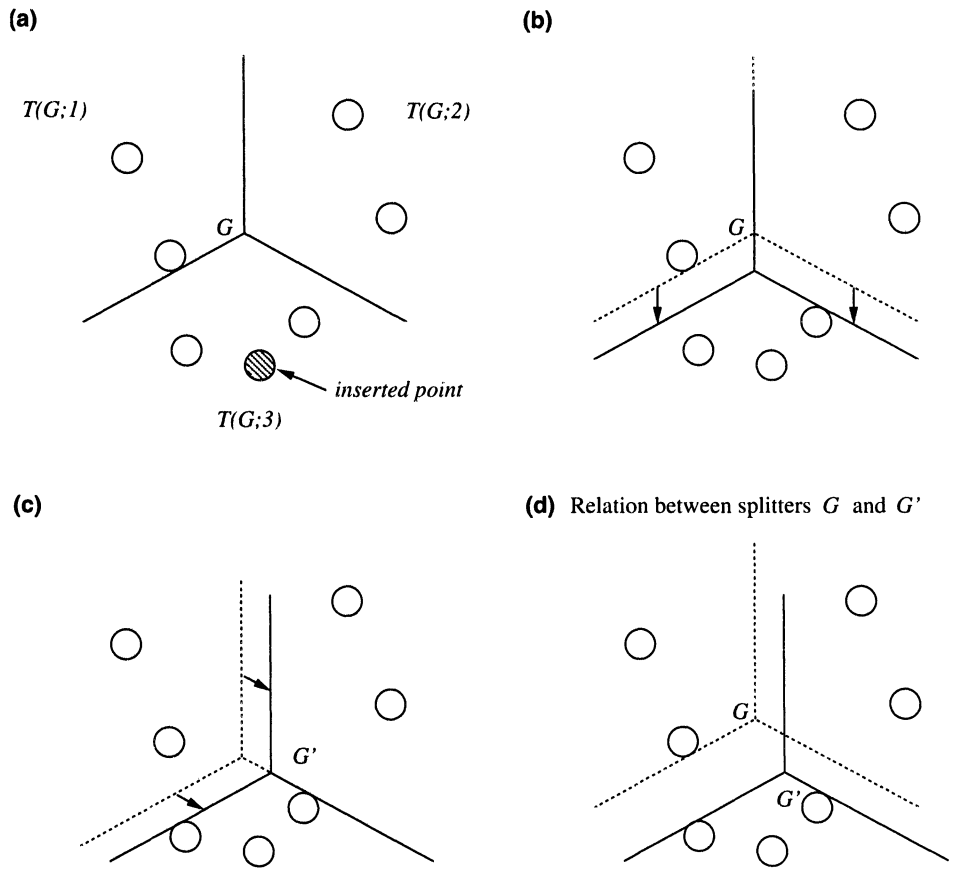


FIG. 3.1. *Update of the splitter.*

Therefore, the algorithm correctly solves the transportation problem.

PROPOSITION 3.5. *The scaling algorithm solves the Hitchcock transportation problem in* $O(k^2 nl \log n)$ *time and* $O(kn)$ *space.*

Proof. Let $G = (g_1, g_2, \ldots, g_k)$ be the current splitter. When a point $p = (p_1, p_2, \ldots, p_k)$ is inserted, we find an index s such that $p_s - g_s = \min_j\{p_j - g_j\}$. Then, it is easy to see that p is contained in $T(G; s)$. This operation requires $O(k)$ time. When UPDATE is called, we must find $k(k-1)$ points $\{q(a, b) \mid a, b = 1, 2, \ldots, k;\ a \neq b\}$. If we provide $k(k-1)$ priority queues, these points can be found in $O(k^2 \log n)$ time. The storage needed for these priority queues is $O(kn)$, and thus is asymptotically not more than the input size. The time complexity for computing the shortest path tree of $\mathcal{D}$ is $O(k^2)$. Thus, the overall complexity for an insertion is $O(k^2 \log n)$. During each stage, at most k points are split. In consequence, the total number of insertions is not more than $n(l+1) + kl$. Thus, the proposition is proven. □

The above algorithm is not strongly polynomial. To make it strongly polynomial, we apply Orlin's contraction technique [16]. We consider a point p in S with a weight ω. Suppose that p is decomposed into $\mathcal{P} = \{p^{(h)} \mid h \subset \{0, 1, \ldots, l\}\}$ of $\tilde{S}$. Instead of adding all portions of p in the procedure HITCHCOCK, we only take the first $\lceil \log_2 8nk \rceil$ largest portions of $\mathcal{P}$ for the input. We contract the other small portions to a point $\mathrm{cont}(p)$.

Suppose we have just inserted all portions of p except $\mathrm{cont}(p)$. The size of each point which is or will be inserted during the current stage is at most $\omega/8nk$. Thus, the total weight

of the remaining points of $\tilde{S}$ is at most $n(1 + 1/2 + 1/4 + \cdots)\omega/8nk < \omega/4k$ after this stage has been completed.

Since at least $(1 - \frac{1}{4nk})\omega$ of the weight of p has been inserted at the end of the current stage, there exists a region (which we denote as the $f(p)$th region) to which at least $\omega/2k$ of the weight of p is assigned. Thus, the portion of p assigned to the $f(p)$th region is greater than twice of the total weight of the remaining points and we obtain the following claim.

CLAIM 3.6. *In the final splitting, at least $\omega/4k$ of the weight of p is assigned to the $f(p)$th region.*

Using the above claim, we modify the algorithm as follows. When the first $\lceil \log 8nk \rceil$ portions of a point p have been inserted (we denote the current grain size by 2^{l-h}), we assign $\mathrm{cont}(p)$ to the $f(p)$th region just after the insertion. Let $w(\mathrm{cont}(p))$ be the weight of the contracted portion. Apparently, $w(\mathrm{cont}(p)) < 2^{l-h}$. The assignment of $\mathrm{cont}(p)$ to $f(p)$ is algorithmically done by simply replacing $\lambda_{f(p)}$ by $\lambda_{f(p)} - w(\mathrm{cont}(p))$. Since the 2-adic expansion of $\lambda_{f(p)}$ is changed, it may happen that the $f(p)$th region overflows by 2^{l-h} after this replacement. In this case, we remove a 2^{l-h} portion of an arbitrary point in the $f(p)$th region, put it in L, and go to step 2 of SPLIT(j, G, W).

Although we have modified the algorithm as above, Properties 1 and 2 both hold. The modified algorithm outputs a splitter of the points, each of which has a weight smaller than the original weight by the weight of the contracted portion. The demands λ_j ($j = 1, 2, \ldots, k$) are also modified during the algorithm. The contracted portion $\mathrm{cont}(p)$ is assigned to $f(p)$ for each point p. It remains to show that the assignment of $\mathrm{cont}(p)$ to $f(p)$ does not contradict to the splitter output by the algorithm, which follows immediately from Claim 3.6. Hence, the algorithm correctly computes the transportation.

We have reduced the number of insertions from $O(nl)$ to $O(n \log nk)$. Note that although we use the 2-adic expansions of ω_i and λ_j, it is easy to see that it suffices to compute their first $O(\log n)$ terms. Hence, we obtain Theorem 3.1.

We remark that the expected performance of the algorithm is much better than its worst case complexity, since the number of updates of the splitter is usually much smaller than that of insertions. An insertion without updating of the splitter is done in $O(k)$ amortized time if we use Fibonacci heaps [9] as priority queues in the algorithm.

4. Randomized algorithms for the c-grained case.

4.1. Simple randomized algorithm for small c. The scaling algorithm solves the Hitchcock transportation problem in $O(k^2 n \log^2 n)$ time. However, in the special case where $\omega_i = 1$ for $i = 1, 2, \ldots, n$, the problem is known to be solvable in $O(kn + k^{2.5} n^{0.5} \log^{1.5} n)$ time by a randomized algorithm [20], [19].

Let us try to use a randomization technique under more relaxed conditions.

DEFINITION 4.1. *A Hitchcock transportation problem is called c-grained for a constant $c \geq 1$ if $1 \leq \omega_i \leq c$ for all $1 \leq i \leq n$. Here, ω_i $(1 \leq i \leq n)$ are real numbers.*

In this section, we show that if $n > c^2 k^4 \log ck$, we can improve the performance of the algorithm by pruning points using random sampling for a c-grained transportation problem.

THEOREM 4.2. *The c-grained Hitchcock transportation problem is solvable in $O(nk + c^{2/3} n^{2/3} k^{10/3} \log^{7/3} n)$ time with a probability larger than $1 - n^{-\gamma}$ for any constant γ.*

In particular, if $n > c^2 k^7 \log^7 ck$, the problem can be solved in the optimal $O(nk)$ time. Note that we can often decompose some heavily weighted points into lighter weighted ones to reduce c. Furthermore, we will see in the next section that it does not matter if there are a small number of points whose weights are less than 1.

Let S_0 be a subset of S consisting of m points chosen uniformly at random from S. Let G be the λ-splitter that we want to find.

LEMMA 4.3. *The total weight of the points of S_0 located in $T(G; j)$ is less than $\frac{m\lambda_j}{n} + \sqrt{rcm\lambda_j/n}$ with a probability larger than $1 - e^{-r/4}$.*

Proof. A random variable X denotes the sum of the weight of points of S_0 located in $T(G; j)$. We define a random variable $X(p)$ as follows. If the point p is in $T(G; j)$, then $X(p) = \omega(p)$ (the weight of the point), otherwise $X(p) = 0$. Thus, the expected value of X is

$$\mathrm{E}[X] = \sum_{p \in S_0} \mathrm{E}[X(p)] = m\left(\frac{1}{n} \sum_{p \in T(G;j)} \omega(p)\right) = \frac{m\lambda_j}{n}.$$

Here, for a c-grained case, the following Chernoff-like bounds hold (see the appendix for proof):

$$(4.1) \quad \begin{cases} \Pr\bigl(X > (1+\delta)\mathrm{E}[X]\bigr) < \exp\{-\mathrm{E}[X]\delta^2/4c\} & (0 < \delta \leq 4.1)\,, \\ \Pr\bigl(X < (1-\delta)\mathrm{E}[X]\bigr) < \exp\{-\mathrm{E}[X]\delta^2/2c\} & (0 < \delta < 1)\,. \end{cases}$$

The lemma is easily derived from them. □

Let $\Lambda = \sum_{j=1}^{k} \lambda_j$ and let W be the total weight of points in S_0. Then, the following corollary holds.

COROLLARY 4.4. *With a probability larger than $1 - e^{-m\Lambda\delta^2/4cn}$,*

$$W > (1-\delta)\frac{m\Lambda}{n}.$$

Here, we choose $\delta = \sqrt{rcn/m\Lambda}$, which gives a probability of $1 - e^{-r/4}$ in the above corollary, and hereafter we consider only the case where $\delta < 1$.

We define a vector $\mu(1) = (\mu(1)_1, \ldots, \mu(1)_k)$ as follows:

$$(4.2) \quad \mu(1)_j = \begin{cases} \dfrac{\lambda_j W}{\Lambda} + \sqrt{\dfrac{rcm\lambda_j}{n}} & \text{for } j \neq 1\,, \\ \dfrac{\lambda_1 W}{\Lambda} - \displaystyle\sum_{i=2}^{k} \sqrt{\dfrac{rcm\lambda_i}{n}} & \text{for } j = 1\,. \end{cases}$$

We calculate the $\mu(1)$-splitter $G(1)$ of S_0, which satisfies the following lemma.

LEMMA 4.5. *The probability that $G(1)$ is located in $T(G; 1)$ is greater than $1-(k-1)e^{-r/4}$.*

Proof. If $G(1)$ is located in the interior of $T(G; j)$ for $j \neq 1$, then all the points in $T(G(1); j)$ are also in $T(G; j)$. Thus, there are more than $\mu(1)_j$ points in $T(G; j)$ in this case, which can occur with a probability less than $e^{-r/4}$, as the previous lemma shows. □

COROLLARY 4.6. *The probability that all the points of S in the interior of $T(G(1); 1)$ lie in the interior of $T(G; 1)$ is greater than $1 - (k-1)e^{-r/4}$.*

For i $(i = 2, 3, \ldots, k)$, we have a slightly different definition of vectors $\mu(i)$:

$$(4.3) \quad \mu(i)_j = \begin{cases} \dfrac{\lambda_j W}{\Lambda} + \Delta_j & \text{for } j \neq i\,. \\ \dfrac{\lambda_i W}{\Lambda} - \displaystyle\sum_{1 \leq \ell \leq k, \ell \neq i} \Delta_\ell & \text{for } j = i\,, \end{cases}$$

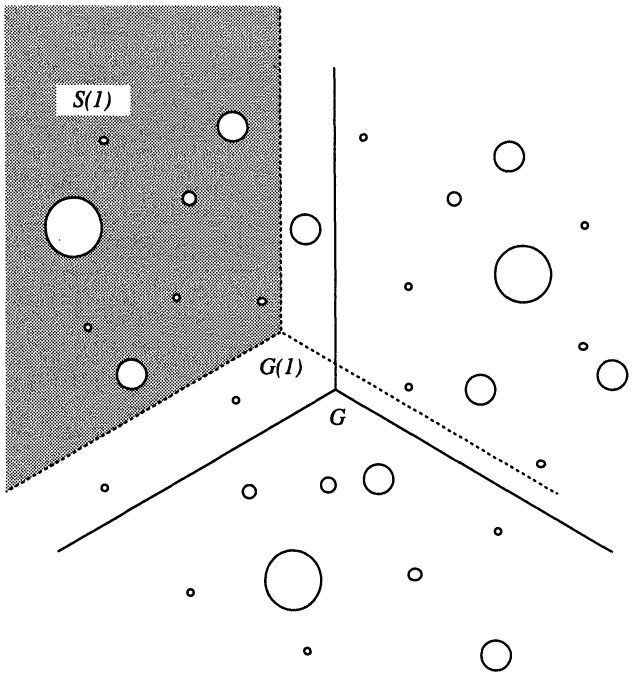


FIG. 4.1. *The set* $S(1)$.

where $\Delta_j = \sqrt{rcm\lambda_j/n} + \epsilon$. Let $G(i)$ be the associated splitter of S_0 with $\mu(i)$. Here, we introduce an arbitrary small real number $\epsilon > 0$ to obtain the following lemma.

LEMMA 4.7. *For all* i $(i = 2, 3, \ldots, k)$, *the splitter* $G(i)$ *is contained in* $T(G(1); i)$.

Proof. Suppose that $G(i) \subset T(G(1); j)$ for some i $(2 \leq i \leq k)$ and $j \neq i$. Obviously, $j \neq 1$ (that is, $T(G(i); 1) \not\subset T(G(1); 1)$), because $\mu(i)_1 > \frac{\lambda_1 W}{\Lambda} > \mu(1)_1$. For $j \neq 1$, since $\mu(i)_j = \mu(1)_j + \epsilon$, the property that $T(G(i); j) \subset T(G(1); j)$ leads to a contradiction. Thus, $G(i) \subset T(G(1); i)$. □

Now, let $S(i)$ $(i = 1, 2, \ldots, k)$ be the set of points of S located in $T(G(i); i)$ (Fig. 4.1) and let ν_i be the total weight of points of $S(i)$. The expected value of ν_1 is

$$\mathrm{E}[\nu_1] = \frac{n}{m}\left(\frac{\lambda_1}{\Lambda}\mathrm{E}[W] - \sum_{i=2}^{k}\sqrt{\frac{rcm\lambda_i}{n}}\right) = \lambda_1 - \frac{n}{m}\sum_{i=2}^{k}\sqrt{\frac{rcm\lambda_i}{n}},$$

and ν_1 is more than

$$\mathrm{E}[\nu_1] - \frac{n}{m}\left(\frac{\lambda_1}{\Lambda}\frac{m\Lambda}{n}\delta + \sqrt{\frac{crm\lambda_1}{n}}\right) \geq \lambda_1 - cn\sqrt{\frac{rk}{m}}, \tag{4.4}$$

with a probability greater than $1 - ke^{-r/4}$ (see Corollaries 4.4 and 4.6). Similarly, we can estimate ν_i for $i = 2, \ldots, k$.

Let us describe a simple randomized algorithm. Choose a sample of size m, set $r = \log k + \gamma \log n$ for a constant γ, and calculate $G(1), \ldots, G(k)$ using the scaling algorithm. For each point p of S, we find a region containing p in the splitting with respect to $G(1)$. If it happens to be in $T(G(1); 1)$, p is a point of $S(1)$. If it is in $T(G(1); i)$ and $i \neq 1$, we check whether it is contained in $T(G(i); i)$. This operation needs $O(k)$ time for each point (from Lemma 4.7).

Now we define a set $\bar{S} = S - \cup_{i=1}^{k} S(i)$ and let $\xi = \lambda - \nu$.

PROPOSITION 4.8. *A ξ-splitter of $\bar{S}$ is a λ-splitter of S with a probability larger than $1 - ke^{-r}$.*

Thus, we can find the λ-splitter of S using the scaling algorithm to compute the ξ-splitter of $\bar{S}$.

Let us analyze the complexity of the algorithm. The probability $1-ke^{-r}$ becomes $1-n^{-\gamma}$, since we choose $r = \log k + \gamma \log n$. Since the weight of points in $\bar{S}$ is $O(ckn\sqrt{rk/m})$, the number of points in $\bar{S}$ is also bounded by $O(ckn\sqrt{rk/m})$. Thus, it takes $O(k^2 \log^2 n(ckn\sqrt{kr/m}))$ time to compute the splitter of $\bar{S}$. On the other hand, it takes $O(k^3 m \log^2 m)$ time for the scaling algorithm to compute k splitters of the sample points. If we set $m = c^{2/3} r^{1/3} k^{4/3} n^{2/3}$, we obtain Theorem 4.2, and for this choice of m, the constraint $\delta < 1$ becomes $c < k^4 \Lambda^3 / n \log^2 n$, which is always satisfied in the c-grained case.

REMARK 4.9. *The above algorithm needs only $O(mk)$ working space. Hence, if the weights can be computed from $O(N)$-space information, the space complexity of the algorithm becomes $O(N+mk)$. For example, N is $O(n)$ in the geometric assignment problem* [2] *where the weight is given by the Euclidean distance between supply and demand points.*

4.2. Iterative randomized algorithm. In the previous randomized algorithm, we essentially reduce the total weight of points from $O(cn)$ (corresponding to S) down to $O(cn\sqrt{rk^3/m})$ (corresponding to $\bar{S}$) with a probability larger than $1 - ke^{-r/4}$, by sampling m points. In the iterative randomized algorithm, we use this sampling method recursively to find a splitter for the point set $\bar{S}$. Furthermore, to simplify the problem, we use the following randomized rounding technique [17].

For a point with real-valued weight $\omega_i = \alpha_i + \beta_i$ (α_i is an integer; $0 \le \beta_i < 1$), we randomly round ω_i to $\bar{\omega}_i$ with the following probability:

$$\begin{cases} \Pr(\bar{\omega}_i = \alpha_i + 1) = \beta_i , \\ \Pr(\bar{\omega}_i = \alpha_i) = 1 - \beta_i . \end{cases}$$

Let $X_i = \bar{\omega}_i - \alpha_i$, $X = \sum_{i=1}^{m} X_i$, and $B = \sum_{i=1}^{m} \beta_i = \mathrm{E}[X]$. Since X can be regarded as a number of successes in a Bernoulli trial, from (4.1) we have

$$\begin{cases} \Pr(X > (1+\delta)B) < e^{-B\delta^2/4} & (0 < \delta \le 4.1) , \\ \Pr(X < (1-\delta)B) < e^{-B\delta^2/2} & (0 < \delta < 1) . \end{cases}$$

By setting $B\delta^2 = r$, we obtain the following lemma (note that $B < m$).

LEMMA 4.10. *With a probability larger than $1 - 2e^{-r/4}$, we have*

$$\sum_{i=1}^{m} \omega_i - \sqrt{rm} < \sum_{i=1}^{m} \bar{\omega}_i < \sum_{i=1}^{m} \omega_i + \sqrt{rm} .$$

Therefore, when we add the remaining $(n-m)$ points, the sum of the weights of each region deviates by $O(n\sqrt{r/m})$ due to the randomized rounding, which is fairly small compared to the term $cn\sqrt{rk/m}$ in (4.4), and we can legitimately neglect this deviation hereafter.

For an integer-valued Hitchcock transportation problem, we modify the sampling method as follows. Each point $p(j)$ ($j = 1, 2, \ldots, n$) with integral weight $\bar{\omega}_j$ will be chosen with a probability proportional to $\bar{\omega}_j$, and when selected, it will contribute to the sampling with a unit amount and $\bar{\omega}_j$ will be reduced by 1. In this case, all the m sample points have a unit weight and the problem for these sample points is essentially a minimum-cost assignment problem and can be solved in $O(km + k^{2.5} m^{0.5} \log^{1.5} m)$ time in the same framework of splitter finding [20].

RANDOM_HITCHCOCK
1. $S = \{\text{input points}\} = \{p(1), \ldots, p(n)\}$;
2. **while** $|S| \geq m$;
   begin
2.1. $S_0 = \emptyset$; (Set of sample points)
2.2. generate a random sequence $\sigma = (\sigma_1, \ldots, \sigma_m)$ according to the rounded weights $\bar{\omega}_i$ $(1 \leq i \leq n)$
2.3. **for** $i = 1$ **to** m;
   begin
2.3.1. add to S_0 a point with the same location as $p(\sigma_i)$ and with a unit weight;
2.3.2. $\bar{\omega}_{\sigma_i} = \bar{\omega}_{\sigma_i} - 1$;
   end;
2.4. **for** $j = 1$ **to** k;
   begin
2.4.1. calculate the $\mu(j)$-splitter $G(j)$ for S_0 deterministically; (see equations (4.2) and (4.3))
2.4.2. $S(j) = \{\text{points of } S \text{ in } T(G(j); j)\}$;
2.4.3. $\nu_j = \text{total weight of points of } S(j)$;
   end;
2.5. $S = S - \bigcup_{j=1}^{k} S(j)$;
2.6. $\lambda = \lambda - \nu$;
   end while;
3. calculate λ-splitter G of S deterministically;
end;

Since the total weight of points whose assignments have not yet been decided is reduced by a factor of $\sqrt{m/rk^3}$ in each iteration, we need $O(\log(cn)/\log(m/rk^3))$ iterations in total. Considering that the time complexity of this algorithm is $O((kn + k^{3.5}m^{0.5}\log^{1.5} n)\log(cn)/\log(m/rk^3) + k^2 m \log^2 n)$ and that $m = \Omega(rk^3)$, we obtain the following theorem.

THEOREM 4.11. *The c-grained Hitchcock transportation problem is solved in*

$$\begin{cases} O\left(\dfrac{kn \log cn}{\log(n/k^4 \log^2 k)}\right) \text{ time}, & \text{when } n > k^4 \log^2 k, \\ O(k^5 \log^2 n \log cn) \text{ time}, & \text{when } n < k^4 \log^2 k, \end{cases}$$

with a probability larger than $1 - n^{-\gamma}$ *for any positive constant* γ.

Proof. Since the number of iterations is $O(\log cn)$ and the probability that we fail to find the splitter is $O(ke^{-r/4}\log cn)$, we choose $r = 4(\gamma \log n + \log k + \log\log cn) = O(\log n)$ to get a success probability of $1 - n^{-\gamma}$, and therefore $m = \Omega(k^3 \log n)$. If we set $m = \Theta(n/k \log k)$ when $n > k^4 \log^2 k$, and $m = \Theta(k^3 \log n)$ when $n < k^4 \log^2 k$, we get the above result. □

COROLLARY 4.12. *If* $n = \Omega(k^{4+\epsilon})$ $(\epsilon > 0)$ *and* $c = \text{poly}(n)$, *then with a high probability, the iterative randomized algorithm solves the Hitchcock transportation problem in* $O(kn)$ *time, which is optimal.*

Both of the randomized algorithms are of Monte Carlo type, since they may output a wrong answer. However, we can easily judge whether the output is a correct splitter or not, so the algorithms can be transformed into Las Vegas type if we repeat them until we get the correct answer.

5. Incremental method. In this section, we consider an incremental version of the transportation problem, where the number of demand points is fixed and the supply points are inserted in a nonincreasing order of weight.

Given a minimum cost transportation on a complete directed bipartite graph with n supply and k demand points, we either increase the supply of a point by ω, or create a new supply point with supply ω. In the latter case, we assume that the cost of the edges between the new supply point and the demand points are also given. The demand of each demand point is increased so that the total demand is increased by ω.

If we solve the minimum cost transportation of the incremented network efficiently, we can design an incremental algorithm for the transportation problem, where the supply points are given in a nonincreasing order of weight and the demands are adjusted according to the total supply.

THEOREM 5.1. *Suppose that G is a ν-splitter of a point set S ($|S| \leq n$). We insert a point p with weight ω. Let λ satisfy $\lambda_j \geq \nu_j$ and $\sum_{j=1}^{k} \lambda_j = \omega + \sum_{j=1}^{k} \nu_j$. Then, if no point of S has a weight less than ω, we can find a λ-splitter in $O(k^3 \log n)$ time, if we permit $O(nk \log n)$ preprocessing time.*

Proof. We first prepare $k(k-1)$ priority queues in $O(nk \log n)$ time. (Note that each point is stored in $k-1$ priority queues.) Each region $T(G; i)$ $(1 \leq i \leq k)$ has generally such $k-1$ interior points that are closest to one of the $k-1$ boundaries of $T(G; i)$. Let us call these points *nearest-boundary* points. Since there is no point in S that has a smaller weight than p, at most $k(k-1)$ nearest-boundary points, which can be found in $O(k^2 \log n)$ time, and at most $k-1$ boundary points will possibly be reassigned to different regions due to the insertion of the point p. To update the assignment of the points, we solve the following minimum-cost flow problem.

The node set is $\{v_1, v_2, \ldots, v_k\}$, where each v_i corresponds to the region $T(G = (g_1, \ldots, g_k); i)$ (or, the ith demand point). For each pair of i and j $(1 \leq i, j \leq k;\ i \neq j)$ suppose that $x = (x_1, \ldots, x_k)$ and $y = (y_1, \ldots, y_k)$ are the closest interior points in $T(G; i)$ and $T(G; j)$, respectively, to the boundary $B_{\{i,j\}}$ between these two regions. Then we have two uncapacitated edges between v_i and v_j: $(v_i, v_j)_1$ with cost $(x_j - x_i) - (g_j - g_i)(> 0)$ and $(v_j, v_i)_1$ with cost $(y_i - y_j) - (g_i - g_j)(> 0)$. Moreover, if there is a boundary point on $B_{\{i,j\}}$, let b_i and b_j be the portion of its weight assigned to $T(G; i)$ and $T(G; j)$, respectively. Then we add two more edges with zero cost but finite capacity: $(v_i, v_j)_2$ with capacity b_1 and $(v_j, v_i)_2$ with capacity b_2. If the inserted point p is in $T(G; \ell)$, the node v_ℓ has a supply $\omega - (\lambda_\ell - \nu_\ell)$ and node v_i $(i \neq \ell)$ has a demand $\lambda_i - \nu_i$. Orlin [16] showed that the minimum-cost flow on a network with N nodes and M edges, where M' of them have finite capacities, can be computed in $O((N + M') \log N(M + N \log N))$ time. Hence, we can find the minimum-cost flow in $O(k^3 \log k)$ time and obtain the theorem. □

In the above formulation of incremental problem, the total supply must be equal to the total demand. However, as shown in the introduction, this condition can be removed. For example, if the total supply is greater than the total demand, we put a virtual demand point of the size of the exceeded supply so that the exceeded supply is transported to it free. If we apply Theorem 5.1 by considering the virtual demand point and the virtual supply point, we can solve the incremental problem under the condition that each demand is increased so that the total increase τ of the demand is at most ω. We can apply the algorithm of Theorem 5.1 with the modification that we either increase the demand of the virtual demand point by $\omega - \tau$ or decrease the supply of the virtual supply point by $\omega - \tau$. The only concern is that the weight of the virtual supply point might be less than ω. In such a case, we remove the virtual supply point and insert it after inserting the current point.

Using the algorithm of Theorem 5.1, we obtain an $O(k^3 n \log n)$ time algorithm for solving a Hitchcock transportation problem by inserting the supply points in the largest-first fashion. Moreover, for the c-grained case, we can design a dynamic algorithm by using Theorem 5.1. When we insert a point with weight a, we decompose it into $\lceil a \rceil$ points with weight less

than or equal to 1, and apply Theorem 5.1. Then a point is inserted in $O(ck^3 \log n)$ time. Furthermore, we can generalize Theorem 4.11 as follows.

COROLLARY 5.2. *Suppose that there are $h = o(n)$ points with weight less than 1 and that the other $n - h$ points satisfy a c-grained condition for $c = \text{poly}(n)$; then the Hitchcock transportation problem can be solved in $O(nk + k^5 \log^3 n + hk^3 \log n)$ time with a probability larger than $1 - n^{-\gamma}$ for any constant γ.*

Proof. First, calculate the splitter for the $n - h$ points by running the procedure RANDOM_HITCHCOCK. Then sort the remaining h points and add them in largest-first fashion. □

Appendix: Proof of inequality (4.1). We estimate the total weight of the sample points that fall in the region $T(G; 1)$ (G is the λ-splitter). For the sake of convenience, we index the n points with integers $1, 2, \ldots, n$ so that $T(G; 1)$ contains points with indices $1, \ldots, n_1$. Let Z_i $(1 \le i \le m)$ be random variables that take on the index of the ith sample point and define X_i and X as follows:

$$X_i = \begin{cases} w_{Z_i} & \text{if } 1 \le Z_i \le n_1 \\ 0 & \text{otherwise} \end{cases} \qquad (1 \le i \le m),$$

$$X = \sum_{i=1}^{m} X_i .$$

The random variable X represents the total weight of the sample points selected from $T(G; 1)$ whose expectation is

$$\begin{aligned} \mathrm{E}[X] = m\mathrm{E}[X_i] &= m \sum_{j=1}^{n_1} w_{Z_i} \Pr(Z_i = j) \\ &= \frac{m}{n} \sum_{j=1}^{n_1} w_j \\ &= \frac{m\lambda_1}{n} \equiv \mu . \end{aligned}$$

We now proceed to prove the analogues of Chernoff's bound.

$$\begin{aligned} \Pr\left(X > (1+\delta)\mu\right) &= \Pr\left(e^{tX} > e^{t(1+\delta)\mu}\right) \quad (t > 0) \\ &< \frac{\mathrm{E}[e^{tX}]}{e^{t(1+\delta)\mu}} \quad \text{(Markov's inequality)} \\ &= \frac{\prod_{i=1}^{m} \mathrm{E}[e^{tX_i}]}{e^{t(1+\delta)\mu}} . \end{aligned}$$

Since we have the constraints $w_j \le c$ $(1 \le j \le n)$,

$$\begin{aligned} \mathrm{E}[e^{tX_i}] &= \sum_{j=1}^{n_1} \frac{1}{n} e^{tw_j} + \frac{n - n_1}{n} \\ &< \frac{\lambda_1}{nc}(e^{tc} - 1) + 1 < \exp\left[\frac{\lambda_1}{nc}(e^{tc} - 1)\right] . \end{aligned}$$

Hence for an arbitrary $t > 0$,

$$\Pr\left(X > (1+\delta)\mu\right) < \frac{\exp\left[\frac{\mu}{c}(e^{tc} - 1)\right]}{e^{t(1+\delta)\mu}} .$$

To minimize the right-hand side of this inequality, we set $e^{tc} = 1 + \delta$, and obtain

$$\Pr\Big(X > (1+\delta)\mu\Big) < \left[\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right]^{\mu/c}.$$

For $0 < \delta \leq 4.1$, this becomes

$$\Pr\Big(X > (1+\delta)\mu\Big) < \exp\left(-\frac{\mu\delta^2}{4c}\right).$$

The other bound can be proved in the same manner.

REFERENCES

[1] R. K. AHUJA, J. B. ORLIN, C. STEIN, AND R. E. TARJAN, *Improved algorithms for bipartite network flow*, SIAM J. Comput., 23 (1994), pp. 906–933.

[2] F. AURENHAMMER, F. HOFFMANN, AND B. ARONOV, *Minkowski-type theorems and least-squares partitioning*, in Proc. 8th ACM Symposium on Computational Geometry, 1992, pp. 350–357.

[3] M. S. BAZARAA AND J. J. JAVIS, *Linear Programming and Network Flows*, John Wiley & Sons, New York, 1977.

[4] E. S. BUFFA AND J. S. DYER, *Management Science/Operations Research*, 2nd ed., John Wiley & Sons, New York, 1981.

[5] V. CHVÁTAL, *Linear Programming*, W. H. Freeman and Company, San Francisco, 1983.

[6] J. G. ECKER AND M. KUPFERSCHMID, *Introduction to Operations Research*, John Wiley & Sons, New York, 1988.

[7] H. EDELSBRUNNER, *Algorithms in Combinatorial Geometry*, ETACS Monograph in Theoretical Computer Science 10, Springer-Verlag, Berlin, 1986.

[8] J. EDMONDS AND R. M. KARP, *Theoretical improvements in algorithmic efficiency for network flow problems*, J. Assoc. Comput. Mach., 19 (1972), pp. 248–264.

[9] M. FREDMAN AND R. TARJAN, *Fibonacci heaps and their uses in improved network optimization algorithms*, J. Assoc. Comput. Mach., 34 (1987), pp. 596–615.

[10] S. FUJISHIGE, *An $o(m^3 \log n)$ capacity-rounding algorithm for the minimum cost circulation problem: A dual framework to tardos's algorithm*, Math. Programming, 35 (1986), pp. 298–309.

[11] Z. GALIL AND E. TARDOS, *An $o(n^2(m+n\log n)\log n)$ min-cost flow algorithm*, in Proc. 27th IEEE FOCS, 1986, pp. 136–146.

[12] D. GOLDFARB AND M. J. TODD, *Linear programming*, Optimization Handbooks in OR and Management Science, Vol. 1, North-Holland, Amsterdam, 1989, pp. 73–170.

[13] T. MATSUI, *Linear time algorithm for the hitchcock transportation problem with a fixed number of supply points*, University of Tokyo, 1991, preprint.

[14] K. MURTY, *Linear and Combinatorial Programming*, John Wiley & Sons, New York, 1976.

[15] K. NUMATA AND T. TOKUYAMA, *Splitting a configuration in a simplex*, Algorithmica, 9 (1993), pp. 649–668.

[16] J. ORLIN, *A faster strongly polynomial minimum cost algorithm*, in Proc. 20th ACM STOC, 1988, pp. 377–387.

[17] P. RAGHAVAN, *Lecture notes on randomized algorithms*, Tech. report RC15340, IBM Research Report, 1990.

[18] E. TARDOS, *A strongly polynomial minimum cost circulation algorithm*, Combinatorica, 5 (1985), pp. 247–255.

[19] T. TOKUYAMA AND J. NAKANO, *Geometric algorithms for the minimum cost assignment problem*, Random Structures Algorithms, to appear.

[20] ———, *Geometric algorithms for the minimum cost assignment problem*, in Proc. 7th ACM Symposium on Computational Geometry, 1991, pp. 262–271.

SIAM J. COMPUT.
Vol. 24, No. 3, pp. 579–597, June 1995

011

APPROXIMATE MAX-FLOW ON SMALL DEPTH NETWORKS*

EDITH COHEN†

Abstract. We consider the maximum flow problem on directed acyclic networks with m edges and *depth* r (length of the longest s-t path). Our main result is a new deterministic algorithm for solving the relaxed problem of computing an s-t flow of value at least $(1-\epsilon)$ of the maximum flow. For instances when r and ϵ^{-1} are small (i.e., $O(\text{polylog}(m))$), this algorithm is in $\mathcal{NC}$ and uses only $O(m)$ processors, which is a significant improvement over existing parallel algorithms. As one consequence, we obtain an $\mathcal{NC}$ $O(m)$ processor algorithm to find a bipartite matching of cardinality $(1-\epsilon)$ of the maximum (for $\epsilon^{-1} = O(\text{polylog}(m))$). We use a novel approach based on path-counts to compute blocking flows in parallel. This approach produces fractional flow even when capacities are integral. For this case we provide a rounding algorithm that is of independent interest. In polylogarithmic time using $O(m)$ processors, the algorithm rounds any fractional flow on a network with integral capacities to an integral flow. The rounding technique extends to networks with costs.

Key words. maximum flow, parallel algorithms, blocking flow, bipartite matching

AMS subject classifications. 90B10, 68Q22, 68Q25, 90C27, 05C85

1. Introduction.

A flow network is modeled by a directed acyclic graph $G = (V, E)$ with capacities $u_e \geq 0$ $(e \in E)$ associated with the edges, a designated *source* vertex $s \in V$, and a designated *sink* vertex $t \in V$. We use the parameter r to denote the *depth* of the network, which we define to be the length of the longest s-t path in G. For a vertex $v \in V$, we denote by $\text{in}(v)$ (resp., $\text{out}(v)$) the set of edges directed into (resp., out of) v. An s-t *flow* f on G is an assignment of flow values f_e $(e \in E)$ to edges such that the capacity constraints are not violated (for all $e \in E$, $0 \leq f_e \leq u_e$), and flow is conserved at every vertex other than the source or the sink, that is, for all $v \in V \setminus \{s, t\}$, $\sum_{e \in \text{in}(v)} f_e = \sum_{e \in \text{out}(v)} f_e$. The *value* of an s-t flow is the amount of excess flow at t (i.e., $\sum_{e \in \text{in}(t)} f_e - \sum_{e \in \text{out}(t)} f_e$). A maximum flow (max-flow) is an s-t flow of maximum value. We use M^* to denote the value of the max-flow.

Finding a max-flow is one of the classic problems in combinatorial optimization. The fastest known sequential algorithms for the problem have time bounds close to $O(mn)$ (see [1], [2], [13]). The known parallel algorithms for the problem are less satisfactory. The max-flow problem with general capacities is known to be $\mathcal{P}$-complete [16] and hence not likely to have an $\mathcal{NC}$ algorithm. The fastest known deterministic parallel algorithm solves the problem (on a CRCW PRAM) in $O(n^2 \log n)$ time using $O(n)$ processors (an algorithm by Shiloach and Vishkin [25] uses $O(n^2)$ space and a later algorithm by Goldberg and Tarjan [13] uses $O(m)$ space). Ahuja and Orlin [1] presented an $O(n^2 \log U \log(m/n))$ time $O(m/n)$ processor EREW PRAM algorithm, where U is the maximum capacity. When the capacities are polynomial in m, the problem seems easier: it is not known to be $\mathcal{P}$-complete, and furthermore, it is in $\mathcal{RNC}$ by a standard logspace reduction to the maximum cardinality bipartite matching problem. The bipartite matching problem was first shown to be in $\mathcal{RNC}$ by Karp, Upfal, and Wigderson [20], who gave an algorithm that uses $O(n^{6.5})$ processors. Later algorithms (see [10], [23]) use fewer processors; the best current bound is $O(nM(n))$ by Galil and Pan [10]. When capacities are in $\{0, 1\}$, the max-flow problem can be solved within the same bounds as maximum cardinality bipartite matching. If the capacities are polynomial in m, an additional factor of $O(m)$ is added to the processor bound. (When capacities are general a factor of $\log U$ is added to the running time, and hence the algorithm is not in $\mathcal{RNC}$.) To summarize, we note that max-flow with polynomial capacities can be solved sequentially using $\tilde{O}(mn)$ work, [1] in parallel in $\tilde{O}(n^2)$ time using $\tilde{O}(n^3)$ work, and in $\mathcal{RNC}$ using expected $\tilde{O}(mnM(n))$ work.

*Received by the editors August 31, 1992; accepted for publication (in revised form) December 29, 1993.
†AT&T Bell Laboratories, Murray Hill, New Jersey 07974 (edith@research.att.com).
[1] We use the notation $\tilde{O}(f) \equiv O(f \text{ polylog } f \text{ polylog } n)$.

We present a max-flow algorithm with the following bounds.

THEOREM 1.1. *A flow of value M such that*

$$M \geq (1-\epsilon)M^*$$

can be computed deterministically on an EREW PRAM in

1. $O(r^3\epsilon^{-3}\log^2 m \log(m\epsilon^{-1}))$ *time using* $O(m/r)$ *processors and* $O(m)$ *space, or*
2. $O(r^2\epsilon^{-2}\log^3 m \log(r\epsilon^{-1})\log(m\epsilon^{-1}))$ *time using* $M(n)$ *processors and* $O(M(n))$ *space.*

Observe that when the depth r and ϵ^{-1} are small (e.g., $O(\text{polylog}(m))$), the resource bounds obtained here are significantly better than known bounds given for general exact max-flow algorithms.

Our max-flow algorithm consists of iterating calls to a subproblem of finding approximate *blocking flows*, and is based on a novel parallel blocking flow algorithm. A blocking flow is an s-t flow which cannot be augmented without pushing flow "backward" through at least one edge. Dinic's algorithm [4] introduced a scheme to solve the max-flow problem by reducing it to blocking flow computations. Dinic's algorithm computes a max-flow in $O(n)$ phases, where each phase amounts to a blocking flow computation on a directed acyclic layered network. The depth of the layered network increases at each phase. The fastest known sequential blocking flow algorithm is due to Goldberg and Tarjan [15] and has an almost optimal time bound of $O(m\log(n^2/m))$. The fastest known (polynomial-work) deterministic parallel algorithm for computing blocking flows was given by Goldberg and Tarjan [14] and runs in $O(n\log n)$ time, using $O(m)$ processors. When capacities are polynomial in m, the problem is in $\mathcal{RNC}$ since this is true for the more general max-flow problem. Unlike the more general $\mathcal{P}$-complete max-flow problem, there is no "hard evidence" that the blocking flow problem is not in $\mathcal{NC}$.

We give an overview of the ideas used and the structure of the paper. In §2 we review commonly used terminology for network flow algorithms, sketch Dinic's max-flow algorithm, and discuss the blocking flow problem. In §3 we examine the use of Dinic's algorithm to obtain near-maximum flows. We show that in order to obtain a flow of value at least $(1-\epsilon)M^*$, it suffices to run Dinic's algorithm for at most $O(r\epsilon^{-1})$ phases (until the length of the layered networks exceeds $O(r\epsilon^{-1})$). Furthermore, we show that when only a near-maximum flow is desired, it is not necessary to find an exact blocking flow at each phase. We define a notion of an approximate blocking flow and prove that we can use a relaxed variant of Dinic's algorithm where exact blocking flow computations are replaced by approximate ones.

In §3 we shall see that the blocking flow instances that arise when running the approximate version of Dinic's algorithm involve networks of depth at most $O(r\epsilon^{-1})$. This fact does not benefit us when considering sequential algorithms since blocking flows can be found almost optimally regardless of the depth of the network [15]. We can gain in the parallel case, however. The worst-case behavior of the $O(n\log n)$ time Goldberg–Tarjan [14] parallel blocking flow algorithm does not improve when instances are limited to networks of small depth. In §4 we present a novel parallel approximate blocking flow algorithm that is significantly more efficient on small depth networks than existing exact blocking flow algorithms. The algorithm uses $O(m)$ processors and is in $\mathcal{NC}$ when $r = O(\text{polylog}\, m)$. If the capacities are polynomial in m, the algorithm computes an exact blocking flow within the same resource bounds.

Our parallel blocking flow algorithm is based on a notion of *path counts* of a network. We define the path count of an acyclic network to be the sum over all s-t paths of the product of all edge capacities along the path. The path count of an edge $e \in E$ is the same sum restricted to paths that contain e. Observe that when the network has unit capacities, the path count reduces to the number of different s-t paths. The algorithm is fairly simple and we sketch it here. We make use of the property that path counts of edges are conserved at all vertices other

than the source and the sink. That is, for each vertex $v \in V \setminus \{s, t\}$, the sum of path counts of edges entering v equals the sum of path counts of edges exiting v. Hence, an assignment of flow values to edges where the values are proportional (with a small enough proportion factor) to the respective path counts constitutes a legal flow. The blocking flow algorithm repeatedly computes the path counts at edges, finds the largest proportional flow augmentation that does not violate the current capacities, and updates the current flow. The following iteration is applied to the network with the recent augmentation being subtracted from the capacities. The algorithm terminates when the current flow is approximately blocking. We will show that the path counts computations, and hence each iteration, can be implemented efficiently in parallel.

The crucial property of the algorithm is that the number of iterations is small (namely, $\tilde{O}(r)$). We give some reasons why this is indeed the case. Consider, for simplicity, a network with unit capacities. A maximal independent set of s-t paths corresponds to a blocking flow. A general approach to find, in parallel, a maximal independent set in a graph is to use iterations, where in each iteration we find some independent set. In the following iteration this independent set and its neighbors are removed from the graph. To obtain a small number of iterations, we need to use independent sets with large neighborhoods. In the blocking flow setting, such an independent set corresponds to a flow augmentation that significantly reduces the path count of the network. The flow augmentation used by the algorithm achieves that by assigning more flow to edges with a large path count (those that have many s-t paths going through them).

The blocking flow algorithm of §4 conceivably terminates with a fractional flow, even when capacities are integral. In §5 we describe an $\mathcal{NC}$ "rounding" algorithm that uses $O(m)$ processors. For a given fractional flow of value M on any network with integral capacities, the algorithm computes an integral flow of value at least $\lceil M - 1/\operatorname{poly}(n) \rceil$. We also describe how to obtain an integral blocking flow by combining the fractional approximate-blocking flow algorithm of §4 with the rounding technique, and how to round a fractional circulation in a network with costs, with at most a small increase in cost. Our rounding algorithm generalizes a similar algorithm for bipartite matching networks given by Goldberg et al. [11].

In §6 we discuss parallel exact max-flow algorithms for zero-one and unit networks. Zero-one networks are networks where all capacities are in $\{0, 1\}$. A unit network is a zero-one network where each vertex has either at most one incident outgoing edge or at most one incident ingoing edge. We give algorithms that use balancing techniques similar to the ones used by Goldberg, Plotkin, and Vaidya in [12], and combine the blocking flow algorithm of §4 and a parallel version of the Ford–Fulkerson algorithm [7]. We obtain sublinear parallel running times that are comparable to the running times given in [12].

Section 7 is concerned with the maximum cardinality bipartite matching problem. By a standard reduction, an instance of the problem can be reduced to an integral max-flow problem on a zero-one network of depth $r = 3$. The corresponding max-flow instance is such that an integral flow of some value M yields a matching of cardinality M in the original problem. Hence, the results of previous sections imply an $O(\log^3 m)$ time $O(m)$ processor deterministic parallel algorithm for finding a bipartite matching of cardinality at least $(1 - \epsilon)$ of the cardinality of the maximum matching when $\epsilon^{-1} = O(1)$. Note that the algorithm remains in $\mathcal{NC}$ and uses $O(m)$ processors even when $\epsilon^{-1} = O(\operatorname{polylog} m)$.

2. Preliminaries. The parallel running times and processor use mentioned in this paper are for the EREW PRAM model. The routines used are limited to standard ones (e.g., list rankings, breadth-first search (BFS), shortest path computations, finding Eulerian partitions) and hence running times on other models can be deduced easily (see, e.g., the survey by Karp and Ramachandran [19]). We use $M(n)$ to denote the best currently known upper bound on the number of operations needed to multiply two $n \times n$ matrices. It is known (see Coppersmith and

Winograd [3]) that $M(n) = o(n^{2.376})$. The quantity $M(n)$ appears in some of our processor bounds for parallel algorithms because of a result of Pan and Reif [24], who had shown that two $n \times n$ matrices can be multiplied in $O(\log n)$ time using $M(n)$ processors, and the fact that the best-known processor bounds for $\mathcal{NC}$ computation of transitive closure, directed s-t path, and BFS tree are obtained using matrix multiplication.

We review some standard terminology associated with maximum flow algorithm and sketch Dinic's algorithm. For further details see [27]. A flow in a network is *blocking* if every path from s to t contains a saturated edge. Consider a current flow f in G and the corresponding residual graph R. The *level* of a vertex $v \in V$ is the length of the shortest path from s to v in R. The *level graph* L for f is the subgraph of R that contains only vertices reachable from s and only edges $e = (u, v)$ such that $\text{level}(v) = \text{level}(u)+1$. The construction of the level graph for a given flow f can be done in $O(m)$ time using breadth-first search. In parallel, the construction can be done in $O(r \log m)$ time using $O(m/r)$ processors, where r is the level of t (using a straightforward parallel implementation of the breadth-first search algorithm).

Dinic's max-flow algorithm [4] maintains a current flow f, initialized as the zero flow, and repeats the following until t is not reachable from s in the residual graph for f.

1. Find a blocking flow f' on the level graph for the current flow f.
2. Replace $f \leftarrow f + f'$.

Dinic showed that the level of t strictly increases in each phase and hence the algorithm terminates after at most $n - 1$ phases.

The complexity of each iteration of Dinic's algorithm is dominated by a blocking flow computation on a layered network. It is not known whether the blocking flow problem on networks of arbitrary depth is either $\mathcal{P}$-complete or is in $\mathcal{NC}$. We sketch an $\mathcal{NC}$ but very inefficient algorithm for finding a blocking flow in networks of a fixed depth r. The algorithm is based on a similar algorithm of Fisher, Goldberg, and Plotkin [6] which was used to show that approximate matching is in $\mathcal{NC}$. In §4 we present a much more efficient algorithm that uses $O(m/r)$ processors and runs in $\tilde{O}(r^2)$ time (alternately, $M(n)$ processors and $\tilde{O}(r)$ time). Note that the latter algorithm is in $\mathcal{NC}$ when $r = O(\text{polylog } m)$.

Consider a network of depth r with $\{0, 1\}$ capacities. A blocking flow can be computed in polylogarithmic time using $n^{O(r)}$ processors as follows. The idea, roughly, is to consider all $O(n^r)$ s-t paths and construct a dependency graph such that every vertex corresponds to an s-t path and two vertices are adjacent if and only if the two corresponding paths share at least one edge. A maximal independent set in the dependency graph corresponds to a set of paths that, when saturated, comprise a blocking flow. The algorithm computes a maximal independent set (this can be done within the stated resource bounds) and obtains a blocking flow. If capacities are integral and polynomial in m, we can use the previous method and replace edge of capacity u_e by u_e parallel edges. Note that the algorithm requires $n^{O(r)}$ processors. When capacities are integral but possibly large, a blocking flow can be obtained by iterating the above algorithm on instances with polynomial capacities $\log(U_f/m)$ times, where U_f is the capacity of the fattest path. In each iteration, we round down the capacities to units of $\lceil U_f/m \rceil$ and truncate large capacities to $m^2\lceil U_f/m \rceil$. As a result, we obtain an instance of the blocking flow problem where all capacities are in the range $\{1..m^2\}$. In the following iteration, the flow values obtained are subtracted from the capacities, and the fattest path with respect to the updated capacities has capacity less than $\lceil U_f/m \rceil$.

3. Obtaining a near-maximum flow. We examine applying variants of Dinic's algorithm to solve the relaxed goal of obtaining a near-maximum flow, when (i) the networks have small depth or (ii) only approximately-blocking flow routines are available.

3.1. Small depth networks. Consider a network G and an s-t flow f. The following proposition states a relation between the value of f and M^* in terms of the depth r and the length of the minimum augmenting path in the residual graph induced by f.

PROPOSITION 3.1. *Consider a network G as above and a flow f of value M where the shortest augmenting path in the residual graph induced by f is of length $\ell \geq r+1$. Under these conditions*

$$M^* \leq \left(\frac{\ell + r}{\ell - r}\right) M .$$

Proof. Without loss of generality, we assume integral flow values and capacities. Consider an integral flow f of value M. Denote by f^* a maximum integral flow with value M^*. There exists a set P of $M^* - M$ augmenting paths of capacity 1 from s to t on the residual graph that (i) augments the flow from f to f^* and (ii) does not use any edge in more than one direction. (Obviously, all paths in P are of length $\geq \ell$.) Note that $M^* = M + |P|$.

For the analysis, we assign vertices to layers labeled $\{0, \ldots, r\}$, where a vertex $v \in V$ is assigned according to the length of the longest path in G from s to v. It is clear that all edges are directed from lower levels to strictly higher levels. We account for a unit flow in an arc ("weight" of the arc) as follows: if it is from layer i to layer j, account for it as $|j - i|$. The total weight of a flow on G equals r times the value of the flow.

Consider a blocking flow with no augmenting path of weight less than ℓ (note that weight ℓ suffices for the result of the proposition; length ℓ is a stronger requirement). For an augmenting path $p \in P$, denote by a_p the weight from the forward edges, and by b_p the weight from the backward edges. Obviously $a_p - b_p = r$, $a_p + b_p \geq \ell$. Note that since $\ell \geq r+1$, we have $b_p \geq 1$.

Intuitively, view a path from P as "exchanging" weight b_p out of the old flow with a larger weight a_p added toward achieving the optimal flow.

Formally, since no two paths from P use an edge in opposite directions we have

$$\sum_{p \in P} b_p \leq rM .$$

From the inequalities $a_p - r = b_p$ and $a_p + b_p \geq \ell$ it follows that for all $p \in P$,

$$b_p \geq (\ell - r)/2 .$$

Combining the two we have

$$|P|(\ell - r)/2 \leq \sum_{p \in P} b_p \leq rM .$$

Hence,

$$|P| \leq \frac{2r}{\ell - r} M .$$

REMARK 3.2. *Proposition* 3.1 *is related to a result of Even and Tarjan* [5] *for zero-one networks which states a relation between the length of the shortest augmenting paths in a residual graph and the value of the remaining flow. It is also related to a result by Fisher, Goldberg, and Plotkin* [6], *who showed that if a matching in a graph is such that there is no augmenting path of length smaller than ℓ then the cardinality of the current matching is at least $(1 - 1/\ell)$ of the maximum cardinality matching. Since a bipartite matching can be expressed*

as an instance of maximum flow on a directed acyclic network of depth 3, *Proposition* 3.1 *generalizes the result of* [6].

COROLLARY 3.3. *The value of any blocking flow on G is at least* $M^*/(2r)$.

Proof. The proof is immediate from the fact that for a blocking flow, $\ell \geq r + 1$.

COROLLARY 3.4. *Consider a network G as above. An s-t flow of value at least* $(1-\epsilon)M^*$ *can be computed in* $O(r\epsilon^{-1}m\log(n^2/m))$ *time. Note that when r and* ϵ *are fixed this bound is almost linear.*

Proof. The stated time bound is obtained by running Dinic's algorithm for $O(r\epsilon^{-1})$ phases. Each phase amounts to a blocking flow computation which can be performed in $O(m\log(n^2/m))$ time by a result of Goldberg and Tarjan [15].

3.2. Using approximately blocking flows. We introduce a notion of an approximate blocking flow and a modified version of Dinic's algorithm which uses an approximate blocking flow in each iteration.

DEFINITION 3.5. *Consider a max-flow instance as above. An* ϵ-blocking flow *is a flow for which all nonsaturated s-t paths contain an edge of capacity not greater than* ϵ. *A* 0*-blocking flow is a blocking flow.*

PROPOSITION 3.6 (modified Dinic algorithm). *Consider a modification of the Dinic algorithm where in each phase we compute an* ϵ*-blocking flow on the current layered graph. In the following phase we construct a layered graph by treating edges of capacity no larger than* ϵ *as having capacity* 0.

1. *At the termination of the kth phase, all augmenting paths of length at most k in the residual graph contain an edge of capacity no larger than* ϵ.
2. *The depth of the layered graph strictly increases from phase to phase.*

Proof. The proof is similar to the correctness proof of Dinic's algorithm [4].

COROLLARY 3.7. *Consider a max-flow instance as above. By running the modified Dinic algorithm until the depth of the layered network exceeds* ℓ (*this happens within* ℓ *phases*) *we can compute a flow of value M such that*

$$M + m\epsilon \geq \left(\frac{\ell - r}{\ell + r}\right) M^*,$$

where M^* *is the value of the max-flow.*

Proof. This follows from Proposition 3.1 and the fact that an ϵ-optimal blocking flow can be augmented to a blocking flow using at most $O(m\epsilon)$ units of flow.

In the following section we present a deterministic parallel algorithm for computing an ϵ-blocking flow. The algorithm is particularly efficient when the depth of the network is small. By combining it with Corollary 3.7 we obtain the bounds stated in Theorem 1.1.

4. Parallel ϵ-blocking flow algorithm. We consider the blocking flow problem on a directed acyclic graph $G = (V, E)$ with capacities $u_e \geq 0$ $(e \in E)$, a designated source vertex $s \in V$, and a designated sink $t \in V$. The parameter r denotes the depth of the network.

We present an algorithm for computing an ϵ-blocking flow in G. We first introduce the concept of path counts.

4.1. The path counts of a network.

DEFINITION 4.1. *Suppose that G is a network as above.*

1. *Denote by* $\mathcal{P}$ *the set of all s-t paths in G. Note that* $|\mathcal{P}| \leq n^r$. *Denote by* $\mathcal{P}_{uw}$ *the set of all paths from* $u \in V$ *to* $w \in V$.
2. *The* path count χ *of the network is*

$$\chi = \sum_{p\in\mathcal{P}} \prod_{e\in p} u_e .$$

When $u_e \in \{0, 1\}$ $(e \in E)$, *then* $\chi = |\mathcal{P}|$.

3. *The* path count of an edge $e \in E$ *is*

$$\chi_e = \sum_{\{p \in \mathcal{P} | e \in p\}} \prod_{e' \in p} u_{e'} .$$

4. *The* path count from $u \in V$ to $w \in V$ *is*

$$\chi^{uw} = \sum_{p \in \mathcal{P}_{uw}} \prod_{e \in p} u_e .$$

The algorithm we introduce here relies on the property that the path counts of edges obey "conservation constraints" at vertices, and hence, so does a flow function proportional to the edges' path counts.

PROPOSITION 4.2. *For all* $v \in V \setminus \{s, t\}$,

$$\sum_{e \in \text{in}(v)} \chi_e = \sum_{e \in \text{out}(v)} \chi_e .$$

Proof. Consider a vertex u. Both quantities above equal the path count of u,

$$\chi_u = \chi^{su} \chi^{ut} = \sum_{\{p \in \mathcal{P} | u \in p\}} \prod_{e \in p} u_e .$$

Complexity of computing the path counts. We suggest two algorithms to compute χ and χ_e for all $e \in E$. This is done by first computing for each vertex $v \in V$, the quantities χ^{sv} and χ^{vt}. It is easy to see that the path count χ_e of an edge $e \in E$, where $e = (u, v)$, equals $\chi_e = u_e \chi^{su} \chi^{vt}$.

The quantities χ^{su} and χ^{ut} can be computed as follows:

1. We use the relations $\chi^{ss} = 1$, $\chi^{tt} = 1$ and for $v \in V$, $\chi^{sv} = \sum_{e=(w,v) \in \text{in}(v)} \chi^{sw} u_e$ and $\chi^{vt} = \sum_{e=(v,w) \in \text{out}(v)} \chi^{wt} u_e$. We do one forward pass starting at s to compute the quantities χ^{sv} $(v \in V)$, and one pass backward starting at t to compute χ^{vt} $(v \in V)$. Each pass consists of phases, where in the ith phase $(0 \le i \le r)$ we scan all vertices such that i is the size of the maximum-size path from s to the vertex (in a backward pass, from the vertex to t). Note that we scan a vertex only if all of its predecessors were scanned previously. This can be done in $O(r \log m)$ time using $O(m/r)$ processors on an EREW PRAM.
2. By using $O(\log r)$ matrix multiplications of $n \times n$ matrices. Pan and Reif [24] have shown that multiplication of two $n \times n$ matrices can be done in $O(\log n)$ time using $\max\{n^2, M(n)\}$ processors. Hence, path counts can be computed in $O(\log r \log n)$ time using $M(n)$ processors on an EREW PRAM.

4.2. The ϵ-blocking flow algorithm. We present an ϵ-blocking flow algorithm. The algorithm consists of iterations, where each iteration augments the flow. We refer to the difference between the initial capacity and the current flow value of an edge e as the *current capacity* of e. The flow on an edge e does not decrease, and hence, the current capacity of e does not increase. For the ith iteration, we denote by u_e^i $(e \in E)$ the current capacities, by χ_e^i $(e \in E)$ the respective path counts, and by $0 \le f_e^i \le u_e^i$ $(e \in E)$ the flow augmentation computed during the iteration. An iteration amounts to first computing the path counts χ_e $(e \in E)$, with respect to the current capacities, and then computing a valid flow augmentation which is proportional to the path counts and is of maximum possible value. Figure 1 depicts an example for the quantities computed in a single iteration of Algorithm 4.3. The figure

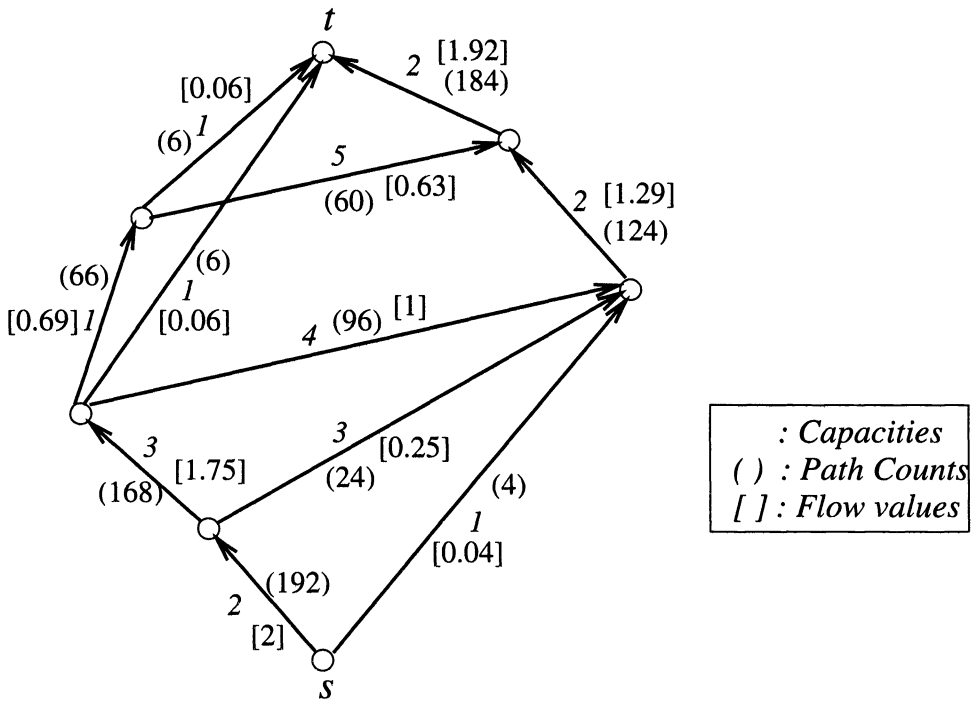


FIG. 1. *Example of a network with path-counts and corresponding flow values.*

shows a capacitated network, the corresponding path counts on the edges, and the resulting flow augmentation.

ALGORITHM 4.3 (Find an ϵ-blocking flow).

1. $i \leftarrow 0$
2. *Set the initial capacities:*

$$\text{for } e \in E,\ u_e^0 \leftarrow \begin{cases} u_e & \text{if } u_e \geq \epsilon, \\ 0 & \text{if } u_e \leq \epsilon, \end{cases}$$

   $E \leftarrow E \setminus \{e \in E | u_e^0 = 0\}$
3. **Repeat:**
   (a) *Compute the path counts* χ_e^i, *for* $e \in E$, *with respect to the capacities* u_e^i. *Let* $\mu_i = \min_{e \in E} u_e^i / \chi_e^i$.
   (b) *For all* $e \in E$, *set* $f_e^i \leftarrow \mu_i \chi_e^i$, $u_e^{i+1} \leftarrow u_e^i - f_e^i$, *if* $u_e^{i+1} \leq \epsilon$ *set* $u_e^{i+1} \leftarrow 0$ $(E \leftarrow E \setminus \{e\})$.
   (c) $i \leftarrow i + 1$

   Until: *t is not reachable from s in E*
4. *Output the flow f, where*

$$\text{for } e \in E,\ f_e \leftarrow \sum_{j=1}^{i-1} f_e^i .$$

4.3. Correctness. For $e \in E$, the quantity $\mu_e^i \equiv u_e^i / \chi_e^i$ is the max-flow value per path count unit we can put through the edge e without violating the capacity u_e^i. The choice of μ_i as the minimum of these quantities guarantees that for all edges $e \in E$, $f_e^i = \mu_i \chi_e^i \leq u_e^i$.

Proposition 4.2 states that the path counts χ_e obey conservation constraints at the vertices. Hence, the same holds for the quantities $f_e^i = \mu_i \chi_e$ ($e \in E$).

We showed that in each iteration, the values f_e^i ($e \in E$) obey both the conservation constraints at the vertices and the current capacity constraints at all edges, and hence, they constitute a valid flow.

It is easy to see that for $i > 0$, at the end of the ith iteration we have

$$\text{for all } e \in E,\ u_e^0 - \epsilon \le u_e^{i+1} + \sum_{i=0}^{i} f_e^i \le u_e^0 .$$

Additionally, the algorithm terminates at the ith iteration if the removal of all edges ($e \in E$) for which $u_e^0 - \sum_{j=0}^{i} f_e^j \le \epsilon$ disconnects s and t. The correctness of the algorithm follows.

PROPOSITION 4.4. *When the algorithm terminates, f_e ($e \in E$) computed in Step 4 is an ϵ-blocking flow.*

Observe that in each iteration, the current capacity of at least one edge is set to 0 (the edge e for which $\mu_i = u_e^i/\chi_e^i$). Hence, the algorithm terminates after at most m iterations. We will prove a much better bound on the number of iterations.

4.4. Bound on the number of iterations.

PROPOSITION 4.5. *For every edge $e \in E$, the quantities μ_e^i ($i \ge 0$) are a nondecreasing function of i, and hence, the same is true for $\mu_i = \min_{e \in E} \mu_e^i$ ($i \ge 0$).*

Proof. The current capacities u_e^i are nonincreasing with respect to i. It follows from the definition of the path count χ_e^i of an edge e that it decreases in at least the same proportion as the current capacity of e (given that all other capacities do not increase). Hence, for every $e \in E$, $u_e^{i+1}/\chi_e^{i+1} \ge u_e^i/\chi_e^i$.

PROPOSITION 4.6.

$$\mu_0^{-1} \le \frac{\left(n \max_{e \in E} u_e^0\right)^r}{n\epsilon} .$$

Proof. In a graph of depth r, there are at most n^{r-1} different paths through some edge e. Each path contributes at most $(\max_{e \in E} u_e^0)^r$ to χ_e^i (for all $e \in E$). To bound the denominator, note that for all $e \in E$, $u_e^0 \ge \epsilon$.

PROPOSITION 4.7. *If for some i,*

$$\mu_i^{-1} \le \epsilon^r / \max_{e \in E} u_e^0 ,$$

then the algorithm terminates at the ith iteration.

Proof. It follows that for all edges $e \in E$,

$$\chi_e^i = u_e^i/\mu_e^i \le \max_{e \in E} u_e^0/\mu^i \le \epsilon^r .$$

Hence, $\max_{e \in E} \chi_e^i \le \epsilon^r$. The latter implies that for every path $p \in \mathcal{P}$, $\prod_{e \in p} u_e^i \le \epsilon^r$, and hence, there exists an edge $e' \in p$ such that $u_{e'}^i \le \epsilon$.

PROPOSITION 4.8. *If*

$$j \ge i + \left\lceil \log\left(\epsilon^{-1} \max_{e \in E} u_e^i\right) \right\rceil ,$$

then $\mu_j/\mu_i \ge 2$.

Proof. Consider the set of edges in a kth iteration

$$E_k = \left\{ e \in E | \mu_k \leq \mu_e^k = u_e^k / \chi_e^k \leq 2\mu_k \right\} .$$

It follows that for every edge $e \in E_k$, the current capacity decreases by a factor of at least $1/2$ during the kth iteration (that is, for every $e \in E_k$ we have $u_e^{k+1} \leq u_e^k/2$). Assume to the contrary that $\mu_j < 2\mu_i$. Consider the edge e such that $\mu_e^j = \mu_j$. Since μ_e^k is nondecreasing with k (see Proposition 4.5), we have $\mu_e^k \leq \mu_e^j \leq 2\mu_i$ (for $i \leq k \leq j$). Hence, since the μ_k's are nondecreasing, we have $\mu_e^k \leq 2\mu_k$ for $k = i, \ldots, j$. Therefore, $e \in E_k$ for $k = i, \ldots, j$. It follows that

$$u_e^j \leq \frac{u_e^i}{2^{(j-i)}} \leq \frac{u_e^i}{\epsilon^{-1} \max_{e \in E} u_e^i} \leq \epsilon .$$

This is a contradiction, since in the jth iteration $e \notin E$.

We obtain the following bound on the number of iterations.

THEOREM 4.9. *The total number of iterations is at most*

$$1 + \left\lceil \log \left(\epsilon^{-1} \max_{e \in E} u_e^0 \right) \right\rceil \left\lceil (r+1) \log \left(n\epsilon^{-1} \max_{e \in E} u_e^0 \right) \right\rceil .$$

Proof. Suppose the algorithm terminates at iteration t. It follows from Propositions 4.6 and 4.7 that

$$\mu_{t-1}/\mu_0 \leq \frac{\max_{e \in E} u_e^0}{n\epsilon} \left(\frac{n \max_{e \in E} u_e^0}{\epsilon} \right)^r .$$

It follows from Proposition 4.8 that the total number of iterations is at most

$$t \leq \left\lceil \log \epsilon^{-1} \max_{e \in E} u_e^0 \right\rceil \lceil \log(\mu_{t-1}/\mu_0) \rceil + 1 .$$

The proof follows by combining the two inequalities.

4.5. Complexity. Denote by U_f the width of the fattest s-t path, i.e.,

$$U_f = \max_{p \in \mathcal{P}} \min_{e \in p} u_e .$$

The quantity U_f is a lower bound on the max-flow and mU_f is an upper bound. The parallel complexity of computing U_f is dominated by the complexity of computing path counts.

We may assume that $\max_{e \in E} u_e \leq mU_f$ and $\min_{e \in E} u_e \geq \epsilon$, since an ϵ-blocking flow when setting all capacities larger than mU_f to mU_f, and capacities smaller than ϵ to 0, is also ϵ-blocking with respect to the original capacities.

PROPOSITION 4.10. *Using Algorithm* 4.3, *an ϵ-blocking flow can be computed in*

1. $O(r^2 \log m \log^2(\epsilon^{-1} m U_f))$ *time using* $O(m/r)$ *processors, or*
2. $O(r \log n \log r \log^2(\epsilon^{-1} m U_f))$ *time using* $M(n)$ *processors.*

Proof. From Theorem 4.9, the number of iterations is bounded by

$$1 + \lceil \log(\epsilon^{-1} m U_f) \rceil \left\lceil (r+1) \log(nm\epsilon^{-1} U_f) \right\rceil = O(r \log^2(\epsilon^{-1} m U_f)) .$$

The complexity of each iteration is dominated by the computation of path counts. Hence, we obtain the stated time bounds by combining the above with the complexity of computing the path counts (see §4.1).

Particularly when $\epsilon = o(U_f/m^k)$, we can obtain a faster running time for finding an ϵ-blocking flow by using the following procedure.

ALGORITHM 4.11 (ϵ-blocking flow when $\epsilon = o(U_f/m)$).
While $U_f > \epsilon$ **Do:**

1. *Compute a* $\max\{U_f/m, \epsilon\}$*-blocking flow (using Algorithm* 4.3.)
2. *Update the capacities (subtract the flow values computed in the previous step).*

It is easy to verify that when using Algorithm 4.11 we obtain the following theorem.

THEOREM 4.12. *An* ϵ*-blocking flow can be computed in*

1. $O(r^2 \log^2 m \log(\epsilon^{-1} m U_f))$ *time using* $O(m/r)$ *processors, or*
2. $O(r \log r \log^3 m \log(\epsilon^{-1} m U_f))$ *time using* $M(n)$ *processors.*

We are now ready to present the proof of Theorem 1.1.

Proof of Theorem 1.1. To obtain these bounds we run the modified Dinic algorithm until the depth of the layered networks exceeds $\ell = 4r\epsilon^{-1}$. In each phase we compute an $\hat{\epsilon}$-blocking flow, where $\hat{\epsilon} = \epsilon U_f^0/2m$ (U_f^0 is the width of the fattest path in G). Denote by M the value of the flow when the modified Dinic algorithm terminates and by M^* the value of the max-flow. We first prove that M and M^* have the desired relation. It follows from the choice of $\hat{\epsilon}$ and ℓ that

$$m\hat{\epsilon} \leq \epsilon M^*/2$$

(since $U_f^0 \leq M^*$), and that

$$(\ell - r)/(\ell + r) \geq (1 - \epsilon/2) .$$

By plugging the above in Corollary 3.7, we deduce that

$$M + \epsilon M^*/2 \geq (1 - \epsilon/2)M^* .$$

Hence, $M \geq (1 - \epsilon)M^*$. We show that the computation can be performed within the desired resource bounds. It is easy to see that all layered networks obtained in the phases of the algorithm have depth at most ℓ, have at most as many nodes and edges as G, and are such that the corresponding quantity U_f is never larger than U_f^0. Hence, it follows from Theorem 4.12 that each phase can be implemented in:

1. $O(\ell^2 \log^2 m \log(\hat{\epsilon}^{-1} m U_f^0))$ time using $O(m/\ell)$ processors, or
2. $O(\ell \log \ell \log^3 m \log(\hat{\epsilon}^{-1} m U_f^0))$ time using $M(n)$ processors.

To conclude note that there are at most ℓ phases.

4.6. Remarks.

REMARK 4.13 (blocking only short paths). *Consider a network G as above. By modifying Algorithm* 4.3 *we can find a depth-d blocking flow, i.e., a flow which blocks all paths of length at most d for some* $d \leq r$. *This is done by using a modified definition of path counts which considers only paths of length at most d. The resource bounds of the modified algorithm are similar, where d replaces r.*

REMARK 4.14 (finding an integral flow). *The max-flow and blocking flow algorithms presented in this section compute a fractional flow, even when capacities are integral. In* §5.3 *we show how to efficiently round any fractional flow to an integral one in* $\mathcal{NC}$ *using* $O(m)$ *processors, and how to obtain an integral* $\lfloor \epsilon \rfloor$*-blocking flow within the resource bounds given for our* ϵ*-blocking flow algorithm.*

5. Rounding fractional flows. We present an $\mathcal{NC}$ algorithm that considers a network G with integral capacities u_e $(e \in E)$ and a fractional s-t flow f_e $(e \in E)$ with value M. The algorithm produces an integral flow $\hat{f}_e$ $(e \in E)$ of value $\hat{M} \in \{\lceil M - 1/\text{poly}(n) \rceil, \lceil M \rceil\}$ such that $\hat{f}_e \in \{\lfloor f_e \rfloor, \lceil f_e \rceil\}$ for $(e \in E)$. We also discuss (i) combining the techniques of this section with the ϵ-blocking flow algorithm of §4 to produce an integral $\lfloor \epsilon \rfloor$-blocking flow and (ii) rounding a fractional circulation with at most a small increase in the cost, in networks where costs are associated with the edges.

It should be noted that this section generalizes an $\mathcal{NC}$ algorithm of Goldberg et al. [11] for obtaining an integral minimum cost flow given a fractional near-minimum cost flow on bipartite matching networks (zero-one networks of depth $r = 3$). Their algorithm uses $O(m)$ processors and is based on Gabow's coloring algorithm [8].

5.1. Main iteration. We start with a subroutine for halving the smallest common denominator of the flow values, assuming it is a power of 2. Suppose $\Delta = 2^{-k}$ for some integer k, and the flow f is such that for all $e \in E$, f_e/Δ is integral. The following procedure computes a flow $\hat{f}$, of value at least as large as the value of f, such that for all $e \in E$, $\hat{f}_e/(2\Delta)$ is integral and for all $e \in E$, $|\hat{f}_e - f_e| \leq \Delta$.

ALGORITHM 5.1 (double unit size).

1. *Mark all edges $e \in E$ for which f_e/Δ is odd.*
   Denote this set of edges by E'.
2. *E' is such that every vertex except for s and t has an even degree (immediate from conservation constraints).*
   Find a Eulerian partition of E', ignoring the directions of the edges. We obtained a collection of cycles and an s-t path. (A Eulerian partition is a collection of edge-disjoint cycles and paths such that all edges incident at even degree vertices belong to cycles and every odd degree vertex is the endpoint of exactly one path.)
3. *Partition the set E' into two sets E^+ (edges labeled $+$) and E^- (edges labeled $-$) as follows:*
   - *If $e \in E'$ belongs to an s-t path in the Eulerian partition do as follows. If e is a forward edge in an s-t traversal of the path, assign $e \in E^+$, otherwise, if e is a backward edge, assign $e \in E^-$.*
   - *Edges that belong to cycles are labeled such that for one of the two traversals, all forward edges are in E^+ and all backward edges are in E^-.*
4. *Compute a flow $\hat{f}_e$ $(e \in E)$ as follows:*
   If $e \in E^+$, $\hat{f}_e \leftarrow f_e + \Delta$.
   If $e \in E^-$, $\hat{f}_e \leftarrow f_e - \Delta$.
   If $e \in E \setminus E'$, $\hat{f}_e \leftarrow f_e$.

The computation is dominated by the construction of the Eulerian partition ($O(\log n)$ time using $O(m)$ processors), and the list ranking operations on segments of total length $|E'| = O(m)$ ($O(\log m)$ time using $O(m)$ processors) (see e.g., [18], [19]). It follows that the subroutine can be implemented on an EREW PRAM to run in $O(\log n)$ time using $O(m)$ processors.

5.2. Rounding algorithm. The following algorithm rounds a flow with general flow values.

ALGORITHM 5.2 (obtain integral flow).

1. *Round up the flow to units of $\Delta_0 = 2^{-5\lceil \log m \rceil}$.*
   Note that this may create excesses and deficits at the vertices. The sum of the absolute values of the excesses and deficits is at most $2m\Delta_0 \leq 2m^{-4} \leq 1/4$. The excesses or

deficits at s and t are determined as if there was an ingoing edge incident at s and an outgoing edge incident at t with flow value of the s-t flow rounded up to units of Δ_0.

2. *For* $k = 0, \ldots, 5\lceil \log m \rceil$ *and* $\Delta = 2^k \Delta_0$:
   If there are excesses and deficits present, run the following modification of Algorithm 5.1. *Otherwise, run the unmodified Algorithm* 5.1.
   Since there are excesses and deficits, the Eulerian partition computed in Step 2 *of Algorithm* 5.1 *may contain paths with end points at vertices other than s and t. The labels of edges along such paths are determined in the same way as cycles, where with respect to some traversal, all backward edges are labeled the same and all forward edges are labeled the same. Use the following rules to determine which of the two possibilities to choose from:*
   - *If a path p is such that an end edge is directed into a vertex with a deficit, put the edge in* E^+. *(If this is true for both end edges of p, arbitrarilychoose one.)*
   - *Otherwise, if p is such that an end edge is directed out of a vertex with an excess, put the edge in* E^+. *(If this is true for both end edges of p, arbitrarily choose one.)*
   - *Otherwise, put one of the end edges of p in* E^-.

Note that (i) at any point, the sum of the excesses and deficits is zero, (ii) the rules to choose the labels guarantee that the sum of the absolute values of excesses and deficits never increases, and (iii) at each vertex, the excess or deficit is an integral number of units of the current value of Δ. Hence, when $\Delta > m^{-4}$, there are no excesses or deficits. Observe that the s-t flow may decrease by at most $2m^{-4}$ during the algorithm, since it may decrease only in iterations where there are excesses or deficits present. It follows that $\hat{M} \geq \lceil M - 2m^{-4} \rceil$.

PROPOSITION 5.3. *The rounding takes* $O(\log^2 m)$ *time using* $O(m)$ *processors, on an EREW PRAM.*

Proof. Step 1 of Algorithm 5.2 can be performed in $O(\log m)$ time using $O(m)$ processors. Step 2 consists of $O(\log m)$ iterations, where each iteration can be performed in $O(\log m)$ time using $O(m)$ processors.

5.3. Computing an integral blocking flow. The rounding algorithm is such that the amount of flow on edges with integral flow is not changed. Hence, when the rounding algorithm is applied to a fractional blocking flow, the result is an integral blocking flow. If the initial fractional flow is ϵ-blocking, the resulting integral flow is $\lceil \epsilon \rceil$-blocking. In this subsection we show how to obtain an $\lfloor \epsilon \rfloor$-blocking flow.

Algorithm 4.3 (ϵ-blocking flow algorithm) produces a fractional ϵ-blocking flow. To obtain an integral $\lfloor \epsilon \rfloor$-blocking flow we use a modified Algorithm 4.3, where at each iteration k, the flow f^k is rounded to an integral flow $\hat{f}^k$ by a rounding algorithm that is presented below. The integral flow $\hat{f}^k$ is used instead of f^k to augment the current flow. Note that the capacities remain integral throughout the execution of the modified Algorithm 4.3.

At each iteration k, the rounding is such that for all $e \in E$, $\hat{f}_e^k \in \{\lceil f_e^k \rceil, \lfloor f_e^k \rfloor\}$. The rounding is done with respect to a parameter η that is determined as follows. Initially, $\eta = \mu_0$. At the beginning of any iteration k, if $2\eta \leq \mu_k$ then we set $\eta = \mu_k$. The rounding will have the following additional property: at least a constant fraction (1/2) of the edges $e \in E$ for which $\mu_e^k \leq 2\eta$ and $u_e^k = 1$ are such that $\hat{f}_e^i = \lceil f_e^i \rceil = 1$. We note that for these edges we have $f_e^i \geq 1/2$. We postpone the discussion of how to achieve this rounding.

We sketch why the bounds of Algorithm 4.3 remain the same. Consider the analysis in §4.4. It is easy to see that Propositions 4.5, 4.6, and 4.7 hold. We prove the correctness of the following modified version of Proposition 4.8. It is easy to see that the remaining part of the

complexity analysis can be carried out with the modified Proposition 4.8 and it yields the same asymptotic bounds.

PROPOSITION 5.4 (modified Proposition 4.8). 1. *When* $\epsilon \geq 1$, *for all iterations* i, j, *if* $j \geq i + \lceil \log_{3/2}(\epsilon^{-1} \max_{e \in E} u_e^i) \rceil$, *then* $\mu_j \geq 2\mu_i$.

2. *When* $\epsilon < 1$, *if iterations* i, j *are such that* $j \geq i + \lceil \log_{3/2}(\max_{e \in E} u_e^i) \rceil + \lceil \log m \rceil + 1$, η *must be updated at least once between the* i*th and* j*th iterations.*

Proof. We first consider the case where $\epsilon \geq 1$. Since the algorithm eliminates edges of capacity ϵ or less, all capacities are at least 2. Consider the kth iteration ($i \leq k \leq j$) and an edge $e \in E_k$ (where E_k is defined as in the proof of Proposition 4.8). We have $f_e^k \geq u_e^k/2$. Hence, $\hat{f}_e^k \geq \lfloor f_e^k \rfloor \geq u_e^k/3$. Therefore, $u_e^{k+1} \leq \lfloor 2u_e^k/3 \rfloor$. The proof follows. We consider the case where $\epsilon < 1$. Assume to the contrary that for some i, j as above, the value of η is not modified. By definition, $\eta \leq \mu_i \leq \cdots \leq \mu_j < 2\eta$. At the kth iteration ($i \leq k \leq j$), consider the set of edges $E_k = \{e \in E | \mu_e^k \leq 2\eta\}$. Since μ_e^k are nondecreasing with k, we have $E_i \supseteq \cdots \supseteq E_j$. Employing the argument used above and the fact that $\eta \leq \mu_k$, we obtain that for all $e \in E_k$, if $u_e^k \geq 2$ then $u_e^{k+1} \leq \lfloor 2u_e^k/3 \rfloor$. Therefore, for $k \geq k' = i + \lceil \log_{3/2}(\max_{e \in E} u_e^i) \rceil$, for all $e \in E_k$, $u_e^k = 1$. It follows from the additional property of the rounding that for $k \geq k'$, $|E_{k+1}| \leq |E_k|/2$. Therefore, $E_j = \emptyset$. This implies that at the jth iteration, for all edges $e \in E$, $\mu_e^j > 2\eta$. This is a contradiction since $\mu_j < 2\eta$ and by definition, there must exist an edge e such that $\mu_e^j = \mu_j$.

What remains is to show how to round the flow such that the resulting integral flow $\hat{f}^k$ has the additional property. It suffices to examine a fractional flow f where edges such that $f_e \geq 1/2$ and $u_e = 1$ are marked. We use a variant of Algorithm 5.2 to round the flow such that at least half of the marked edges are rounded up. The variant runs Algorithm 5.2 as stated, except for the last iteration where $\Delta = 1/2$. The choice of Δ_0 ensures us that the sum of absolute values of excesses and deficit is less than 1/4. Hence, at the iteration where $\Delta = 1/2$ there are no excesses or deficits. Therefore, the Eulerian partition consists of a collection of cycles and possibly an s-t path. It is easy to see that at the iteration where $\Delta = 1/2$, the flow on each marked edge e is either $\lceil f_e \rceil$ or $\lceil f_e \rceil - 1/2$. Observe that Step 2 of Algorithm 5.1 allows us, for each cycle or path, two possible choices of labels of edges that do not violate the conservation constraints. We remark that for the s-t path, one of the choices of labels decreases the s-t flow, but for the purpose of rounding the blocking flow, we allow the net flow to decrease. For each cycle and for the s-t path, we choose the labels that maximize the number of marked edges which are selected to be in E^+.

It follows that using the integral version of the ϵ-blocking flow algorithm, we can compute an exact integral blocking flow when the capacities are integral and polynomial in m, in the following EREW PRAM resource bounds:

1. $O(r^2 \log^3 m)$ time using $O(m/r)$ processors, or
2. $O(r \log r \log^4 m)$ time using $M(n)$ processors.

5.4. Rounding a flow in a network with costs. Suppose that we are given a fractional circulation f_e ($e \in E$), possibly with integral excesses and deficits at the vertices (supplies and demands), on a network with integral capacities and real costs c_e ($e \in E$) associated with the edges. We modify Algorithm 5.2 so that it produces an integral flow $\hat{f}_e$ with the properties $\hat{f}_e \in \{\lfloor f_e \rfloor, \lceil f_e \rceil\}$ ($e \in E$) and

$$\sum_{e \in E} c_e \hat{f}_e \leq 2mn\Delta_0 \max_{e \in E} |c_e| + \sum_{e \in E} c_e f_e ,$$

where Δ_0 is some inverse power of 2 (typically, $\log \Delta_0^{-1} \geq 5\lceil \log m \rceil$). The modified algorithm runs in time $O(\log m \log \Delta_0^{-1})$ and uses $O(m)$ processors. The algorithm uses the input value

of Δ_0. Only the excesses and deficits introduced by the rounding are being eliminated. (In other words, the excess/deficit at a vertex is the excess/deficit reduced by the initial integral demand/supply value). In each iteration, we do as follows. For each cycle or path c, there are two ways to label edges with E^+ or E^-. The choice such that $\sum_{e\in p\cap E^+} c_e \leq \sum_{e\in p\cap E^-} c_e$ guarantees that the total cost does not increase as a result of changing the flow on these edges. For cycles, we always choose the labels such that the cost does not increase. Because of the initial rounding, there may be paths with end points at vertices with excesses or deficits. For such paths, if both choices of labels preserve the sum of absolute values of deficits and excesses, we choose the labeling that does not increase the cost. Otherwise, we choose the labeling that decreases the sum of absolute values of deficits and excesses. Note, however, that the cost may increase as a result of the change. Initially, the sum of absolute values of excesses and deficits is at most $2m\Delta_0$. Each unit of excess or deficit may be routed once on a cost-increasing path. The size of each such path is at most n (we may assume that the Eulerian partitions consist of simple paths and cycles, although this assumption is not needed in previous subsections). Hence, the total increase in cost resulting from excess/deficit decreasing paths is bounded by $2mn\Delta_0 \max_{e\in E} |c_e|$.

When

$$\Delta_0 = 2^{-5\lceil \log m\rceil},$$

the running time is $O(\log^2 m)$ and the resulting integral flow is such that

$$\sum_{e\in E} c_e \hat{f}_e \leq 2m^{-4} n \max_{e\in E} |c_e| + \sum_{e\in E} c_e f_e .$$

6. Zero-one and unit networks. Zero-one networks are such that all capacities are in $\{0, 1\}$. A unit network is a zero-one network where in addition, each vertex has either a single incident outgoing edge or a single incident ingoing edge. This section is concerned with max-flow computation on such networks. Even and Tarjan [5] showed that on zero-one (resp., unit) networks, Dinic's algorithm terminates within $O(\min\{n^{2/3}, \sqrt{m}\})$ (resp., $O(\sqrt{n})$) phases. The reason is that if the residual network is such that all augmenting paths are longer than ℓ, the value of the max-flow is bounded by m/ℓ (resp., n/ℓ). Also note that for any integral flow, the residual networks are of the same form (zero-one or unit, resp.). Hence, after $O(\sqrt{m})$ (resp., $O(\sqrt{n})$) phases, the value of the max-flow in the residual network is at most $O(\sqrt{m})$ (resp., $O(\sqrt{n})$).

For such networks we can guarantee that if all augmenting paths are long, the value of the max-flow is small. We obtain parallel max-flow algorithms as follows. We apply Dinic's algorithm for some number of phases, using the integral version of the algorithm of §4 to compute blocking flows. We stop Dinic's algorithm when the depth of the layered network becomes large. To obtain a max-flow, we apply a parallel version of the Ford–Fulkerson max-flow algorithm to the residual network. The Ford–Fulkerson algorithm repeatedly finds and saturates augmenting paths until no augmenting path exists. Since the value of the remaining flow is small, the Ford–Fulkerson algorithm does not perform many iterations. A similar balancing between parallel versions of the Ford–Fulkerson algorithm and the Goldberg–Tarjan push-relabel algorithm [13] was previously used by Goldberg, Plotkin, and Vaidya [12] to obtain sublinear time algorithms for such networks.

For the first part (Dinic's algorithm) of the algorithm we use the ϵ-blocking flow algorithm of §4. In §5.3 we showed that when the capacities are integral of size polynomial in m (as is the case for zero-one networks), the algorithm can be modified to find an exact integral blocking flow. For the second part (Ford–Fulkerson's algorithm), we use the fact that a single augmenting path, if one exists, can be found in $O(\log^3 n)$ time with $M(n)$ processors (using

parallel BFS) and hence, a flow of value f^* can be found in $O(f^* \log^3 n)$ time using $M(n)$ processors. By balancing the two parts of the algorithm we determine the number of iterations used and obtain the following bounds. Note that similar bounds for unit and zero-one networks were previously obtained in [12].

PROPOSITION 6.1. *A maximum flow can be computed on an $M(n)$ processor EREW PRAM in:*

1. $O(n^{2/3} \log^{3+1/3} n)$ *time for unit networks,*
2. $O(m^{2/3} \log^{3+1/3} n)$ *time for zero-one networks,* ($\tilde{O}((\sum_{e \in E} u_e)^{2/3})$ *for general networks with integral capacities).*

Proof. The running times are obtained by first running Dinic's algorithm, using the blocking flow algorithm of §4, until the depth of the layered network exceeds some parameter ℓ. When capacities are small, this first part of the algorithm takes $O(\ell^2 \log \ell \log^3 n)$ parallel time, using $M(n)$ processors (see Proposition 4.10). (Note that we may assume that capacities are integral and at most polynomial in n.) The second part consists of running a parallel version of the Ford–Fulkerson algorithm. The running time is $O(f^* \log^3 n)$, using $M(n)$ processors, where f^* is the value of the remaining flow. For part 6.1 we choose $\ell = (n/\log n)^{1/3}$ and for part 6.1 we choose $\ell = (m/\log n)^{1/3}$.

7. Approximate bipartite matching in parallel. A matching in a graph is a set of edges such that no two edges are incident at the same vertex. The maximum cardinality bipartite matching (MCBM) problem is to find, in a bipartite graph, a matching of maximum size. The problem can be stated as an integral max-flow problem on a unit network of depth 3. The corresponding network is such that the set of saturated edges of any integral flow turns out to be a matching of cardinality which equals the value of that flow. Hence, MCBM can be solved using Dinic's algorithm in $O(m\sqrt{n})$ time [5] (see also [17] for an algorithm stated in terms of alternating paths). A blocking flow computation on a unit network can be done in $O(m)$ time [5], and hence, it follows from Corollary 3.4 that an approximate matching can be found in $O(\epsilon^{-1}m)$ time.

As for the parallel complexity of the MCBM problem, it is still not known whether the problem is in $\mathcal{NC}$. It is known, however, to be in $\mathcal{RNC}$. The first $\mathcal{RNC}^3$ algorithm was obtained by Karp, Upfal, and Wigderson [20]. Their algorithm uses $O(n^{6.5})$ processors. A later $\mathcal{RNC}^2$ algorithm by Mulmuley, Vazirani, and Vazirani [23] uses $O(n^{3.5}m)$ processors. The best known $\mathcal{RNC}$ processor bound of $O(nM(n))$ (in $\mathcal{RNC}^3$) was achieved by Galil and Pan [10]. Lev, Pippenger, and Valiant [21] gave an $\mathcal{NC}$ algorithm for the special case of regular bipartite graphs, and Miller and Naor [22] gave an $\mathcal{NC}$ algorithm for planar bipartite graphs. An algorithm by Goldberg et al. [12] solves the MCBM problem in $O(n^{2/3} \log^3 m)$ time using $M(n)$ processors. An interior point-based algorithm by Goldberg et al. [11] solves the problem in $\tilde{O}(\sqrt{m})$ time using $O(m^3)$ processors.

In [6], Fisher, Goldberg, and Plotkin gave an $\mathcal{NC}$ algorithm which finds a matching in a graph which is at least $(1-\epsilon)$ of the maximum cardinality matching. They first showed that if a matching is such that there are no augmenting paths of length smaller than ℓ, then the matching is of cardinality at least $(1-1/\ell)$ of maximum. Their algorithm is roughly based on examining all paths of length at most ℓ, and hence uses $n^{O(\ell)}$ processors. As a corollary of Theorem 1.1, we obtain an $\mathcal{NC}$ algorithm which finds a bipartite matching of cardinality $(1 - 1/\operatorname{polylog} n)$ of the maximum matching and uses only a linear $O(m)$ number of processors. A similar result was obtained very recently by Spencer [26], who used a modification of the parallel exact matching algorithm of Goldberg, Plotkin, and Vaidya [12]. The results mentioned above are summarized in Table 1.

THEOREM 7.1. *Bipartite matching of cardinality $(1-\epsilon)$ of maximum can be computed in:*

1. $O(\epsilon^{-3} \log^2 m \log(m\epsilon^{-1}))$ *time using $O(m)$ processors, or*
2. $O(\epsilon^{-2} \log^3 m \log \epsilon^{-1} \log(m\epsilon^{-1}))$ *time using $O(M(n))$ processors.*

TABLE 1
Parallel maximum cardinality bipartite matching results.

Algorithm	*Time*	*Processors*
Deterministic, approximation to $(1-\epsilon)$		
Fisher, Goldberg, Plotkin [6]	$O(\log^3 m)$	$O(n^{2\epsilon^{-1}})$
	$O(\epsilon^{-1}m)$	1
This paper	$O(\epsilon^{-3}\log^2 m \log(m\epsilon^{-1}))$	$O(m)$
	$O(\epsilon^{-2}\log^3 m \log \epsilon^{-1} \log(m\epsilon^{-1}))$	$O(M(n))$
Deterministic		
Hopcroft, Karp [17]	$O(m\sqrt{n})$	1
Goldberg et al. [11]	$\tilde{O}(\sqrt{m})$	$O(m^3)$
Goldberg, Plotkin, Vaidya [12]	$O(n^{2/3}\log^3 m)$	$O(M(n))$
Gabow, Tarjan [9]	$O(n\log^2 n)$	$O(m/(n^{1/2}\log n))$
Shiloach, Vishkin [25]	$O(n^{3/2}\log n)$	$O(m/n)$
Randomized		
Karp, Upfal, Wigderson [20]	$O(\log^3 m)$	$O(n^{6.5})$
Mulmuley, Vazirani, Vazirani [23]	$O(\log^2 m)$	$O(n^{3.5}m)$
Galil, Pan [10]	$O(\log^3 n)$	$O(nM(n))$

Proof. The algorithm is as follows. First we translate the bipartite matching instance to an s-t max-flow problem on a unit network of depth $r = 3$. The max-flow on this network equals the maximum cardinality of a matching in the original problem. An integral flow corresponds to a matching. We first find a fractional flow that approximates the max-flow to within a factor of $(1-\epsilon)$. This can be done within the time bound stated in Theorem 1.1. The finishing step amounts to rounding the flow to an integral flow of at least the same value. This was previously done by Goldberg et al. in [11], who gave an $O(\log^2 n)$ time algorithm which uses $O(m)$ processors.

8. Open problems. To conclude, we discuss some issues and open problems that arise from this work.

Achieving better bounds. We presented an $O(M(n))$ processors $\tilde{O}(r)$ time algorithm for approximate blocking flow (exact for polynomial capacities). It is still open whether we can improve on this, that is, find an (approximate) blocking flow in $O(r^{1-\epsilon})$ deterministic parallel time for some $\epsilon > 0$. Such an algorithm would have interesting implications, one of which is that a flow with no augmenting paths shorter than r could be found in $O(r^{2-\epsilon})$ deterministic parallel time. (We remark that computing such a flow within the latter bounds might be an easier task than the first one.) The latter bound, even for the special case of unit networks, would yield a faster time bound than that currently known for computing maximum cardinality bipartite matching deterministically, in parallel, using polynomial work.

Extension to minimum cost flow. We ask whether our results can be extended to finding near-minimum cost flow more efficiently on small depth networks, or more specifically, to minimum-weight bipartite matching. One problem is that the augmenting path argument in the proof of Proposition 3.1 does not seem to carry over for minimum cost flows.

Weighted path counts. In this paper we introduced the notion of path counts and used path counts weighted by capacities to compute a flow augmentation in each iteration of the blocking flow algorithm. We ask whether we can benefit by considering path counts weighted by other "costs" associated with the edges, which could possibly be dependent on the capacities and other parameters. Weighted path counts keep the properties of being efficiently computable in

parallel and being conserved at vertices. Consider, for simplicity, a zero-one network of depth r. Let $\boldsymbol{c} = \{c_e | e \in E\}$ be nonnegative weights on the edges, and denote by χ_e $(e \in E)$ the weighted path counts (relative to the weights c_e). Each such system of weights, corresponds to a flow $f(\boldsymbol{c})$ where

$$f_e(\boldsymbol{c}) = \frac{\chi_e}{\max_{e' \in E} \chi_{e'}} \quad (e \in E) .$$

Denote the value of $f(\boldsymbol{c})$ by $M(\boldsymbol{c})$. We ask how close to the max-flow M^* is the value of the max-flow that has the form $f(\boldsymbol{c})$ for some nonnegative set of costs $\boldsymbol{c}$. For unit networks we have $M^* = \max_{\boldsymbol{c}} M(\boldsymbol{c})$. We suggest an approach for an algorithm that directly approximates the max-flow without breaking the computation to blocking flow phases. The approach is to start with some initial costs and to keep updating them until $M(\boldsymbol{c})$ is close to M^* for the current weights $\boldsymbol{c}$.

Acknowledgments. The author would like to thank Andrew Goldberg, David Johnson, Seffi Naor, Tomek Radzik, Peter Shor, Vijay Vazirani, and Alex Wang for relevant discussions and pointers to the literature, and the anonymous referee for careful reading and many helpful comments.

REFERENCES

[1] R. K. AHUJA AND J. B. ORLIN, *A fast and simple algorithm for the maximum flow problem*, Oper. Res., 37 (1989), pp. 748–759.

[2] R. K. AHUJA, J. B. ORLIN, AND R. E. TARJAN, *Improved time bounds for the maximum flow problem*, SIAM J. Comput., 18 (1989), pp. 939–954.

[3] D. COPPERSMITH AND S. WINOGRAD, *Matrix multiplication via arithmetic progressions*, J. Symb. Comput., 9 (1990), pp. 251–280.

[4] E. A. DINIC, *Algorithm for solution of a problem of maximum flow in networks with power estimation*, Soviet Math. Dokl., 11 (1970), pp. 1277–1280.

[5] S. EVEN AND R. E. TARJAN, *Network flow and testing graph connectivity*, SIAM J. Comput., 4 (1975), pp. 507–518.

[6] T. FISCHER, A. V. GOLDBERG, AND S. A. PLOTKIN, *Approximating matching in parallel*, Inform. Process. Lett., 46 (1993).

[7] L. R. FORD JR. AND D. R. FULKERSON, *Flows in networks*, Princeton University Press, Princeton, NJ, 1962.

[8] H. N. GABOW, *Using Euler partitions to edge-color bipartite multi-graphs*, Int. J. Comput. Inform. Sci., 5 (1976), pp. 345–355.

[9] H. N. GABOW AND R. E. TARJAN, *Almost-optimum speed-ups of algorithms for bipartite matching and related problems*, in Proc. 20th Annual ACM Symposium on Theory of Computing, ACM, 1988, pp. 514–527.

[10] Z. GALIL AND V. PAN, *Improved processor bounds for combinatorial problems in RNC*, Combinatorica, 8 (1988), pp. 189–200.

[11] A. V. GOLDBERG, S. A. PLOTKIN, D. B. SHMOYS, AND É. TARDOS, *Interior point methods in parallel computation*, in Proc. 30th IEEE Annual Symposium on Foundations of Computer Science, IEEE, 1989, pp. 350–355.

[12] A. V. GOLDBERG, S. A. PLOTKIN, AND P. M. VAIDYA, *Sublinear time parallel algorithms for matching and related problems*, in Proc. 29th IEEE Annual Symposium on Foundations of Computer Science, IEEE, 1988, pp. 174–185.

[13] A. V. GOLDBERG AND R. TARJAN, *A new approach to the maximum flow problem*, J. Assoc. Comput. Mach., 35 (1988), pp. 921–940.

[14] ———, *A parallel algorithm for finding a blocking flow in an acyclic network*, Inform. Process. Lett., 31 (1989), pp. 265–271.

[15] ———, *Finding minimum cost circulations by successive approximation*, Math. Oper. Res., 15 (1990), pp. 430–466.

[16] L. M. GOLDSCHLAGER, R. A. SHAW, AND J. STAPLES, *The maximum flow problem is logspace complete for P*, Theoret. Comput. Sci., 21 (1982), pp. 105–111.

[17] J. HOPCROFT AND R. M. KARP, *An $n^{5/2}$ algorithm for maximum matching in bipartite graphs*, SIAM J. Comput., 2 (1973), pp. 225–231.

[18] J. JÁJÁ, *Parallel algorithms*, Addison-Wesley Publishing Co., Reading, MA, 1992.

[19] R. M. KARP AND V. RAMACHANDRAN, *Parallel algorithms for shared-memory machines*, in Handbook of Theoretical Computer Science, Vol. A, J. van Leeuwenm, ed., MIT Press/Elsevier, Cambridge, MA, 1991, pp. 869–941.

[20] R. M. KARP, E. UPFAL, AND A. WIGDERSON, *Constructing a perfect matching is in random NC*, Combinatorica, 6 (1986), pp. 35–48.

[21] G. F. LEV, N. PIPPENGER, AND L. G. VALIANT, *A fast parallel algorithm for routing in permutation networks*, IEEE Trans. Comput., C-30 (1981), pp. 93–100.

[22] G. L. MILLER AND J. NAOR, *Flow in planar graphs with multiple sources and sinks*, in Proc. 30th IEEE Annual Symposium on Foundations of Computer Science, IEEE, 1989, pp. 112–117.

[23] K. MULMULEY, U. V. VAZIRANI, AND V. V. VAZIRANI, *Matching is as easy as matrix inversion*, Combinatorica, 7 (1987), pp. 105–113.

[24] V. PAN AND J. REIF, *Efficient parallel solution of linear systems*, in Proc. 17th Annual ACM Symposium on Theory of Computing, ACM, 1985, pp. 143–152.

[25] Y. SHILOACH AND U. VISHKIN, *An $O(n^2 \log n)$ parallel max-flow algorithm*, J. Algorithms, 3 (1982), pp. 128–146.

[26] T. H. SPENCER, *Parallel approximate matching*, Tech. Report UNO-CS-TR-92-5, University of Nebraska, Omaha, NE, 1992.

[27] R. E. TARJAN, *Data structures and network algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

SIAM J. COMPUT.
Vol. 24, No. 3, pp. 598–620, June 1995

012

A FAST APPROXIMATION ALGORITHM FOR COMPUTING THE FREQUENCIES OF SUBGRAPHS IN A GIVEN GRAPH*

RICHARD A. DUKE†, HANNO LEFMANN‡, AND VOJTĚCH RÖDL§

Abstract. In this paper we give an algorithm which, given a labeled graph on n vertices and a list of all labeled graphs on k vertices, provides for each graph H of this list an approximation to the number of induced copies of H in G with total error small. This algorithm has running time $O(n^{1/\log\log n} \cdot M(n))$, where $M(n)$ is the time needed to square an n by n matrix with 0, 1-entries over the integers. The main tool in designing this algorithm is a variant of the regularity lemma of Szemerédi.

Key words. counting subgraphs, regularity lemma

AMS subject classifications. 68C25, 05C99

1. Introduction.

Given a graph G on n vertices and a positive integer k, the following decision problem arises in a natural way: determine whether G contains a subgraph isomorphic to the complete graph K_k on k vertices.

The question of whether the complexity of this problem is $o(n^k)$ was raised by L. Lovász and others. J. Nešetřil and S. Poljak [NP] provided an algorithm which decides whether G contains a copy of K_k in time $O(M(n^{k/3}))$, where $M(N) = O(N^{2.376})$ is the time requried to multiply two N-by-N matrices with 0, 1-entries over the integers (see [CW]), and noted that F. K. Chung and R. Karp had obtained similar results. It is easy to see, as mentioned in [NP], that a method for deciding whether G contains a copy of K_k leads to an algorithm of the same complexity for determining whether G contains a copy of any other fixed graph H on k vertices as an induced subgraph.

These decision methods could serve as the basis for an algorithm to determine the number of copies of K_k, or the number of induced copies of some other graph H on k vertices, in G, but the resulting algorithm would have essentially the same complexity as that of checking each of the $\binom{n}{k} = O(n^k)$ k-vertex subgraphs of G.

Other authors have considered a list whose entries record the numbers of occurences in a graph G of all possible graphs on k vertices. Frank [F] studied the information which could be derived from such a list for $k = 3$. Similar investigations were done for signed stochastic graphs by Frank and Harary [FH] to measure balance in empirical networks.

Erdös, Lovász, and Spencer [ELS] studied the geometric properties of the vectors whose components are the relative frequencies with which all graphs on k or fewer vertices occur in individual larger graphs.

Our objective in this paper is to show that for sufficiently large n and appropriate values of k, there exists an efficient algorithm which yields an approximation to the number of copies of each graph on k vertices in a given graph on n vertices, and does so with small total error. More specifically, given a labeled graph on n vertices and a list of all labeled graphs on k vertices, this algorithm yields, for each graph H_i in the list, an approximation to the number h_i of induced subgraphs of G which are isomorphic to H_i, with total error at most the average value of the h_i. This algorithm runs in $O(n^2(M(n))$ sequential time.

To describe this result more precisely we need some terminology. Let $G = (V, E)$ be a labeled graph on n vertices whose vertex set $V = \{v_1, v_2, \ldots, v_n\}$ is ordered by $v_1 < v_2 <$

*Received by the editors April 19, 1993; accepted for publication (in revised form) January 11, 1994.

†Georgia Institute of Technology, School of Mathematics, Atlanta, Georgia 30332.

‡Universität Dortmund, Fachbereich Informatik, LS II D-44221 Dortmund, Germany. Part of this author's research was done at the University of Idaho, Department of Mathematics and Statistics, Moscow, Idaho 83843.

§Emory University, Department of Mathematics and Computer Science, Atlanta, Georgia 30322. The research of this author was supported in part by National Science Foundation grant DMS-9401559.

$\cdots < v_n$. Let the set $W = \{w_1, w_2, \ldots, w_k\}$, $k \leq n$, be ordered by $w_1 < w_2 < \cdots < w_k$. We say that a graph H with vertex set W is *order isomorphic* to an induced subgraph H' of G if there exists an isomorphism $\phi : H \to H'$ with the property that for each i and j, $1 \leq i, j \leq k$, $w_i < w_j$ implies $\phi(w_i) < \phi(w_j)$. Let $H_1, H_2, \ldots, H_t$, where $t = 2^{\binom{k}{2}}$, be a list of all labeled graphs on the set W and let $\sigma_k(G) = (h_1, h_2, \ldots, h_t)$ be the t-dimensional vector in which for each i, $1 \leq i \leq t$, h_i is the number of induced subgraphs of G to which H_i is order isomorphic. We call a vector $\bar{\sigma}_k = (\bar{h}_1, \bar{h}_2, \ldots, \bar{h}_t)$, $t = 2^{\binom{k}{2}}$, a *ρ–approximation* to $\sigma_k(G)$ if $|h_i - \bar{h}_i| \leq \rho\binom{n}{k}$ for each i, $1 \leq i \leq t$.

Using this notation our main result can be formulated as follows, where all logarithms are in base 2.

THEOREM 1.1. *Let c be a constant, $0 < c \leq 1$, and n be an integer for which $\log\log n \leq \sqrt{c \log n}$. There is an algorithm which, given a labeled graph on n vertices and an ordering of its vertices and given a list of all labeled graphs on an ordered set of k vertices, $3 \leq k \leq \sqrt{c \log n}$, yields a $2k(k\epsilon)^{1/2}$-approximation to $\sigma_k(G)$ in $O(2^{\binom{k}{2}} n^{(11/6)c} M(n))$ sequential time, where $\epsilon = (2^{16} k^2 \log\log n / c \log n)^{1/21}$ and $M(n)$ is the time required to multiply two n-by-n matrices with 0, 1–entries over the integers.*

To obtain a ρ–approximation to $\sigma_k(G)$, say for $\rho \leq 1$, we need only ensure that $2k(k\epsilon)^{1/2} \leq \rho$. Thus a simple calculation yields the following consequence of this theorem.

COROLLARY 1.2. *Let ρ and c be constants, $0 \leq \rho, c \leq 1$, and k and n be integers satisfying $3 \leq k \leq (\rho^{42} c \log n / 2^{58} \log\log n)^{1/65}$. Then there is a polynomial time algorithm which, given a labeled, ordered graph G on n vertices, n sufficiently large, and a list of all labeled graphs on an ordered set of k vertices, finds a ρ–approximation to $\sigma_k(G)$.*

In particular, for $3 \leq k \leq \sqrt{\frac{1}{100}\log\log n}$, $c = \frac{1}{2}$, and ϵ as in the statement of the theorem, we have that $\log\log n \leq \sqrt{\frac{1}{2}\log n}$ is satisfied and $2k(k\epsilon)^{1/2} \leq 4^{-\binom{k}{2}}$, which ensures an efficient algorithm yielding an approximation $\bar{\sigma}_k(G)$ in which the sum of the errors in all of its terms is at most the average entry size of $\sigma_k(G)$.

COROLLARY 1.3. *There is an algorithm whose input is a labeled, ordered graph on n vertices and a list of all labeled graphs on an ordered set of k vertices, where $3 \leq k \leq \sqrt{\frac{1}{100}\log\log n}$, and whose output is an approximation $\bar{\sigma}_k = (\bar{h}_1, \bar{h}_2, \ldots, \bar{h}_t)$, $t = 2^{\binom{k}{2}}$, to the vector $\sigma_k(G) = (h_1, h_2, \ldots, h_t)$ with the property that*

$$\sum_{1 \leq i \leq t} |h_i - \bar{h}_i| \leq \frac{1}{t} \sum_{1 \leq i \leq t} h_i.$$

This algorithm runs in $O(n^{1/\log\log n} \cdot M(n))$ sequential time.

The proof of Theorem 1.1 makes use of a variant of the regularity lemma of Szemerédi [Sz]. Given a graph G, we partition its vertex set into classes and for a subset of k of these classes, say $V_1, V_2, \ldots, V_k$, we consider the set of k-tuples in $V_1 \times V_2 \times \cdots \times V_k$. We first establish the existence of a type of "regular" partition of such a collection of k-tuples. Next we describe an efficient algorithm for finding a regular partition, cf. [ADLRY] for related results. (A lemma similar to the existence portion of these results has already been considered by Eaton and one of the authors [ER].)

To state these results precisely we need some additional definitions. Let $G = (V, E)$ be a graph and (A, B) a pair of disjoint, nonempty subsets of V. The *density* of the pair (A, B) is given by

$$d(A, B) = \frac{e(A, B)}{|A||B|}.$$

where $e(A, B)$ denotes the number of edges in E joining vertices of A to vertices of B. We call a pair (A, B) *ϵ-regular* if for each choice of $X \subseteq A$ and $Y \subseteq B$ with $|X| \geq \epsilon|A|$ and $|Y| \geq \epsilon|B|$ we have

$$|d(A, B) - d(X, Y)| < \epsilon.$$

Given a k-partite graph $G = (V, E)$, $V = \cup_{1 \leq i \leq k} V_i$, $|V_i| = N$, $i = 1, 2, \ldots, k$, we will consider a partition $\mathcal{K}$ of the *set* $V_1 \times V_2 \times \cdots \times V_k$ where each partition class is of the form $K = W_1 \times W_2 \times \cdots \times W_k$, $W_i \subseteq V_i$, $i = 1, 2, \ldots, k$. We will call such classes *cylinders* and sometimes write $K = V_1(K) \times V_2(K) \times \cdots \times V_k(K)$. We say that the cylinder $K = W_1 \times W_2 \times \cdots \times W_k$ is *ϵ-regular* if the subgraph $G(K)$ of G induced on the set $\cup_{1 \leq i \leq k} W_i$ is such that all $\binom{k}{2}$ of the pairs (W_i, W_j), $1 \leq i < j \leq k$, are ϵ-regular. We say that the partition $\mathcal{K}$ is ϵ-regular if all but ϵN^k of the k-tuples $(v_1, v_2, \ldots, v_k)$, $v_i \in V_i$, $i = 1, 2, \ldots, k$, are contained in ϵ-regular cylinders.

Our algorithm for finding such an ϵ-regular partition $\mathcal{K}$ proceeds by starting with the partition consisting of a single cylinder and refining the partition at hand until ϵ-regularity is established. This calls for efficient means of determining that a given pair (W_i, W_j) is either ϵ-regular or possesses a pair of subsets $X \subseteq W_i$, $Y \subseteq W_j$ whose density is far enough from that of (W_i, W_j) to be used in choosing a refinement. That such a method exists is the result of the following lemma.

LEMMA 1.4. *There exists an algorithm which, given a constant ϵ, $0 < \epsilon < 1$, a graph $G = (V, E)$ and a pair (A, B) of nonempty, disjoint subsets of V with $|A|, |B| \leq N$ and $|A| \geq (2/\epsilon)^5$, will in $O(M(N))$ time verify that the pair (A, B) is ϵ-regular or will find two subsets $X \subseteq A$ and $Y \subseteq B$ with $|X| \geq \delta|A|$ and $|Y| \geq \delta|B|$ such that*

$$|d(A, B) - d(X, Y)| \geq \delta,$$

where $\delta = \epsilon^5/16$.

The proof of the existence, for a given k-partite graph, of an ϵ-regular partition of the k-tuples into cylinders and the proof of Lemma 1.4 are given in §2. Together these lead to the following corollary.

COROLLARY 1.5. *Let $G = (V, E)$ be a k-partite graph with $V = \cup_{1 \leq i \leq k} V_i$, $|V_i| = N$, $i = 1, 2, \ldots, k$. For every $\epsilon > 0$ there exists an ϵ-regular partition $\mathcal{K}$ of $V_1 \times V_2 \times \cdots \times V_k$ into s cylinders such that*

i) *$s \leq 4^p$, where $p \leq \binom{k}{2}\epsilon^{-5}$, and*

ii) *$|V_i(K)| \geq \epsilon^p N$ for each $K \in \mathcal{K}$ and $i = 1, 2, \ldots, k$.*

Furthermore, if $\delta^{q+1}N \geq 2$, where $\delta = \epsilon^5/16$ and $q = \binom{k}{2}/\epsilon\delta^4$, an ϵ-regular partition for G with at most 4^q cylinders can be found in $O(\binom{k}{2}4^{q+1}M(N))$ sequential time.

The proof of Theorem 1.1 is based on this algorithm for finding ϵ-regular partitions and one additional result which we now describe. Given a k-partite graph $G = (V, E)$, $V = \cup_{1 \leq i \leq k} V_i$, we set $d_{i,j}$ equal to the density of the pair (V_i, V_j) for each i and j, $1 \leq i < j \leq k$. Let $H = (W, F)$ be a graph whose k-element vertex set $W = \{w_1, w_2, \ldots, w_k\}$ is ordered by $w_1 < w_2 < \cdots < w_k$. We say that an induced subgraph H' of G is *partite isomorphic* to H if $H' = (W', F')$, where $W' = \{w_1', w_2', \ldots, w_k'\}$, $w_i' \in V_i$, $i = 1, 2, \ldots, k$, and the mapping $w_i \mapsto w_i'$ is an isomorphism. For each such choice of G and H and each i and j, $1 \leq i < j \leq k$, set

$$\hat{d}_{i,j} = \begin{cases} d_{i,j} & \text{if } (w_i, w_j) \in F, \\ 1 - d_{i,j} & \text{otherwise.} \end{cases}$$

Let $f_<(H, G)$ denote the number of induced subgraphs of G which are partite isomorphic to H. The following result is proved in §3.

LEMMA 1.6. *Let $G = (V, E)$, $V = \cup_{1\le i\le k} V_i$, be a k-partite graph and δ a positive constant.*

If for each i and j, $1 \le i < j \le k$, the pair (V_i, V_j) is ϵ-regular, where $\epsilon = \delta^2/k^3$, then we have

$$\left| f_<(H, G) - \prod_{1\le i<j\le k} \hat{d}_{i,j} \prod_{1\le i\le k} |V_i| \right| \le \delta \prod_{1\le i\le k} |V_i|.$$

(Similar results concerning the number of copies of a given graph H in a "uniform" graph G have also been used in [R], [FR], and, more recently, in [SS].)

The proof of Theorem 1.1, making use of Corollary 1.5 and Lemma 1.6, is given in §4.

2. The algorithm for finding a regular partition. First we establish the existence of an ϵ-regular partition of the type described in the previous section.

PROPOSITION 2.1. *Let $G = (V, E)$ be a k-partite graph with $V = \cup_{1\le i\le k} V_i$, $|V_i| = N$, $i = 1, 2, \ldots, k$. For every $\epsilon > 0$ there exists an ϵ-regular partition $\mathcal{K}$ of $V_1 \times V_2 \times \cdots \times V_k$ into s cylinders such that*

i) *$s \le 4^p$, where $p \le \binom{k}{2}\epsilon^{-5}$, and*

ii) *$|V_i(K)| \ge \epsilon^p N$ for each $K \in \mathcal{K}$ and $i = 1, 2, \ldots, k$.*

Proof. We shall make use of the following fact, which we state without proof.

Fact. Let ϵ be a positive constant and γ_i, d_i, $0 < \gamma_i, d_i < 1$, $i = 1, 2, 3, 4$, be real numbers such that

i) $d = \sum_{1\le i\le 4} \gamma_i d_i$,

ii) $\sum_{1\le i\le 4} \gamma_i = 1$, and

iii) $|d_1 - d| \ge \epsilon$.

Then

$$\sum_{1\le i\le 4} \gamma_i d_i^2 \ge d^2 + \epsilon^2\left(\frac{\gamma_1}{1-\gamma_1}\right).$$

Much as in the proof Szemerédi's regularity lemma given in [Sz] we proceed inductively, forming a sequence of partitions, each a refinement of its predecessor. For a cylinder $K = V_1(K) \times V_2(K) \times \cdots \times V_k(K)$ and i, j, $1 \le i < j \le k$, we let $d_{i,j}(K)$ denote the density of the bipartite graph induced on the set $V_i(K) \cup V_j(K)$. That is,

$$d_{ij}(K) = \frac{|E \cap V_i(K) \times V_j(K)|}{|V_i(K)||V_j(K)|}.$$

Set $\mathcal{K}^0 = \{K_1\}$, where $K_1 = V_1 \times V_2 \times \cdots \times V_k$. If $\mathcal{K}^0$ is ϵ-regular; we are done. Suppose, therefore, that it is not. This means that for some i and j, $1 \le i < j \le k$, there are sets $W_i \subseteq V_i$ and $W_j \subseteq V_j$ such that $|W_i| \ge \epsilon|V_i|$, $|W_j| \ge \epsilon|V_j|$, and

$$\left| d_{ij}(K_1) - \frac{|E \cap (W_i \times W_j)|}{|W_i||W_j|} \right| \ge \epsilon.$$

We may further assume that $|W_i| \le (1-\epsilon)|V_i|$ and $|W_j| \le (1-\epsilon)|V_j|$ and, without loss of generality, that $i = 1$ and $j = 2$.

We consider a new partition $\mathcal{K}'$ consisting of four cylinders

$$K_{1,1} = W_1 \times W_2 \times V_3 \times \cdots \times V_k,$$
$$K_{1,2} = W_1 \times (V_2 \setminus W_2) \times V_3 \times \cdots \times V_k,$$
$$K_{1,3} = (V_1 \setminus W_1) \times W_2 \times V_3 \times \cdots \times V_k,$$

and

$$K_{1,4} = (V_1 \setminus W_1) \times (V_2 \setminus W_2) \times V_3 \times \cdots \times V_k.$$

For $\mathcal{K}'$ and each later partition we define an *index* as follows. If $\mathcal{K} = K_1 \cup K_2 \cup \cdots \cup K_r$, then $\mathrm{ind}_{\mathcal{K}} = \sum_{1\le s\le r} \mu(K_s) \sum_{1\le i,j\le k} d_{ij}^2(K_s)$, where

$$\mu(K_s) = \frac{|K_s|}{N^k}.$$

Assume that after $m-1$ steps we have constructed a partition $\mathcal{K}^{m-1}$ which is not ϵ-regular. This means that at least ϵN^k k–tuples in $V_1 \times V_2 \times \cdots \times V_k$ are in ϵ-irregular cylinders. Let $\mathcal{J} \subseteq \mathcal{K}^{m-1}$ be the set of all irregular cylinders of $\mathcal{K}^{m-1}$. We replace each $K_i \in \mathcal{J}$ with four new cylinders $K_{i,1}, K_{i,2}, K_{i,3}, K_{i,4}$ as described above. Set

$$\mathcal{J}' = \bigcup_{K_i \in \mathcal{J}} \{K_{i,1}, K_{i,2}, K_{i,3}, K_{i,4}\},$$

and

$$\mathcal{K}^m = (\mathcal{K}^{m-1} - \mathcal{J}) \cup \mathcal{J}'.$$

We will compare the indices of the partitions $\mathcal{K}^{m-1}$ and $\mathcal{K}^m$. Clearly we have

$$\begin{aligned} \mathrm{ind}_{\mathcal{K}^m} - \mathrm{ind}_{\mathcal{K}^{m-1}} &= \sum_{K_s \in \mathcal{J}} \left\{ \sum_{1\le \ell\le 4} \mu(K_{s,\ell}) \sum_{1\le i,j\le k} d_{ij}^2(K_{s,\ell}) - \mu(K_s) \sum_{1\le i,j\le k} d_{ij}^2(K_s) \right\} \\ &= \sum_{K_s \in \mathcal{J}} \mu(K_s)\sigma(K_s), \end{aligned} \tag{1}$$

where

$$\begin{aligned} \sigma(K_s) &= \sum_{1\le \ell\le 4} \frac{\mu(K_{s,\ell})}{\mu(K_s)} \sum_{1\le i,j\le k} d_{ij}^2(K_{s,\ell}) - \sum_{1\le i,j\le k} d_{ij}^2(K_s) \\ &= \sum_{1\le i,j\le k} \left\{ \sum_{1\le \ell\le 4} \frac{\mu(K_{s,\ell})}{\mu(K_s)} d_{ij}^2(K_{s,\ell}) - d_{ij}^2(K_s) \right\}. \end{aligned} \tag{2}$$

Set $\phi(K_{s,\ell}) = \mu(K_{s,\ell})/\mu(K_s)$. It is easy to see that for each i and j, $1 \le i, j \le k$,

$$d_{ij}(K_s) = \sum_{1\le \ell\le 4} \phi(K_{s,\ell}) d_{ij}(K_{s,\ell}), \tag{3}$$

where

$$\sum_{1\le \ell\le 4} \phi(K_{s,\ell}) = 1. \tag{4}$$

Thus applying Proposition 2.1 with $\epsilon = 0$ we obtain

$$\sum_{1\le \ell\le 4} \frac{\mu(K_{s,\ell})}{\mu(K_s)} d_{ij}^2(K_{s,\ell}) - d_{ij}^2(K_s) \ge 0. \tag{5}$$

For each $K_s \in \mathcal{J}$ and each i and j, $1 \le i, j \le k$. Moreover, due to the fact that $K_s \in \mathcal{J}$ there exist i_0, j_0, $1 \le i_0 < j_0 \le 1$, and some $\ell \in \{1, 2, 3, 4\}$ such that

$$|d_{i_0 j_0}(K_{s,\ell}) - d_{i_0 j_0}(K_s)| \ge \epsilon. \tag{6}$$

Due to the construction of the cylinders $K_{s,\ell}$,

$$\phi(K_{s,\ell}) \ge \epsilon^2 \tag{7}$$

for each ℓ, $\ell = 1, 2, 3, 4$. Combining the fact stated at the beginning with (3), (4), (6), and (7) we have

$$\sum_{1 \le \ell \le 4} \frac{\mu(K_{s,\ell})}{\mu(K_s)} d^2_{i_0, j_0}(K_{s,\ell}) - d^2_{i_0, j_0}(K_s) \ge \epsilon^2 \left(\frac{\epsilon^2}{1 - \epsilon^2} \right). \tag{8}$$

Thus (2), (5), and (8) yield

$$\sigma(K_s) \ge \frac{\epsilon^4}{1 - \epsilon^2}, \tag{9}$$

for each $K_s \in \mathcal{J}$. Since $\sum_{K_s \in \mathcal{J}} \mu(K_s) \ge \epsilon$, we infer from (1) and (9) that

$$\operatorname{ind}_{\mathcal{K}^m} \ge \operatorname{ind}_{\mathcal{K}^{m-1}} + \sum_{K_s \in \mathcal{J}} \mu(K_s) \frac{\epsilon^4}{1 - \epsilon^2} > \operatorname{ind}_{\mathcal{K}^{m-1}} + \epsilon^5. \tag{10}$$

Since it is clear that $\operatorname{ind}_{\mathcal{K}} \le \binom{k}{2}$ for any partition $\mathcal{K}$, this means that for some $p \le \binom{k}{2}\epsilon^{-5}$ the partition $\mathcal{K}^p$ is ϵ–regular.

Next we turn to the proof of Lemma 1.4. First, however, we consider several additional propositions which will be used in that proof. If Y is a subset of the vertices of a graph $G = (V, E)$ and x is a vertex of G, $x \notin Y$, we will use $\deg_Y(x)$ to denote the degree of x in Y, that is, the number of members of Y which are adjacent to x in G.

PROPOSITION 2.2. *Let $G = (V, E)$ be a graph with (A, B) an ϵ-regular pair of subsets of V, $\epsilon > 0$. Suppose that $Y \subseteq B$ with $|Y| \ge \epsilon|B|$ and set $d(A, B) = \rho$. Then we have*

$$|\{x \in A \mid \deg_Y(x) > (\rho - \epsilon)|Y|\}| > (1 - \epsilon)|A|.$$

Proof. Let $X = \{x \in A \mid \deg_Y(x) \le (\rho - \epsilon)|Y|\}$. If $|X| \ge \epsilon|A|$, then

$$d(X, Y) \le \frac{|X|(\rho - \epsilon)|Y|}{|X||Y|} = \rho - \epsilon,$$

contradicting the ϵ-regularity of the pair (A, B).

Since similar reasoning can be used for the set of vertices $x \in A$ with $\deg_Y(x) \ge (\rho + \epsilon)|Y|$, we have the following result, where $Y = B$.

PROPOSITION 2.3. *Let $G = (V, E)$ be a graph with (A, B) an ϵ-regular pair of subsets of V, $\epsilon > 0$. Set $d(A, B) = \rho$. Then we have*

$$|\{x \in A \mid (\rho - \epsilon)|B| < \deg_B(x) < (\rho + \epsilon)|B|\}| > (1 - 2\epsilon)|A|.$$

If x and x' are two vertices of the graph $G = (V, E)$ and Y is a subset of V disjoint from $\{x, x'\}$, then by the *codegree of x and x' in Y*, written $\deg_Y(x, x')$, we mean the number of vertices in Y which are adjacent to *both* x and x' in G. (Note that $\deg_Y(x, x')$ does not in

general coincide with the density $d(\{x, x'\}, Y)$.) The following proposition gives information about codegrees in an ϵ-regular pair.

PROPOSITION 2.4. *Let $G = (V, E)$ be a graph and (A, B) an ϵ-regular pair of disjoint subsets of V, $\epsilon > 0$, having density $d(A, B) = \rho$, where $\rho|B| \geq 1$. Let D be the collection of all pairs $\{x, x'\}$ of vertices of A for which*

i) $(\rho - \epsilon)|B| < \deg_B(x), \deg_B(x') < (\rho + \epsilon)|B|$, *and*

ii) $\deg_B(x, x') < (\rho + \epsilon)^2|B|$.

Then we have

$$]D] > \frac{1}{2}(1 - 5\epsilon)|A|^2.$$

Proof. Let $Z = \{x \in A | (\rho - \epsilon)|B| < \deg_B(x) < (\rho + \epsilon)|B|\}$. By Proposition 2.3 we have $|Z| > (1 - 2\epsilon)|A|$. Let x be an element of Z. We claim that there exist at least $(1 - 3\epsilon)|A|$ vertices $x' \in Z\backslash\{x\}$ such that the pair $\{x, x'\}$ is in D. If not, then the set $X = \{x' \in A | \deg_B(x, x') \geq (\rho + \epsilon)^2|B|\}$ is such that $|X| \geq \epsilon|A|$. In this case we could let Y be a subset of B which contains all of the neighbors of x in B and for which $\epsilon|B| \leq |Y| \leq (\rho + \epsilon)|B|$. Then the density of the pair (X, Y) would satisfy

$$d(X, Y) \geq \frac{|X|(\rho + \epsilon)^2|B|}{|X||Y|} \geq \rho + \epsilon,$$

contradicting the ϵ-regularity of (A, B). It follows that

$$|D| \geq \frac{1}{2}(1 - 2\epsilon)(1 - 3\epsilon)|A|^2 > \frac{1}{2}(1 - 5\epsilon)|A|^2.$$

Remarks. For an ϵ-regular pair (A, B) whose density satisfies $d(A, B) = \rho \geq 2\epsilon$ we have $\deg_B(x) \geq \epsilon|B|$ in the above argument. In this case we could omit the condition $\rho|B| \geq 1$, and could show the existence of at least $\frac{1}{2}(1 - 6\epsilon)|A|^2$ pairs $\{x, x'\}$ of vertices of A satisfying the more symmetric conditions

i) $(\rho - \epsilon)|B| < \deg_B(x), \deg_B(x') < (\rho + \epsilon)|B|$, and

ii) $(\rho - \epsilon)^2|B| < \deg_B(x, x') < (\rho + \epsilon)^2|B|$.

Our next result provides a partial converse to Proposition 2.4.

PROPOSITION 2.5. *Let ϵ be a constant, $0 < \epsilon < 1$. Let $G = (V, E)$ be a graph with (A, B) a pair of disjoint, nonempty subsets of V with $|A| \geq \frac{2}{\epsilon}$. Set $d(A, B) = \rho$. Let D be the collection of all pairs $\{x, x'\}$ of vertices of A for which*

i) $\deg_B(x), \deg_B(x') > (\rho - \epsilon)|B|$, *and*

ii) $\deg_B(x, x') < (\rho + \epsilon)^2|B|$.

Then if $|D| > \frac{1}{2}(1 - 5\epsilon)|A|^2$, the pair (A, B) is δ-regular, where $\delta = (16\epsilon)^{1/5}$.

Proof. We may assume that $d(A, B) = \rho > 0$, since otherwise there is nothing to prove. Let $\delta = (16\epsilon)^{1/5}$ and set with forsight $\lambda = \frac{1-\rho}{\rho}$. Let $A = \{a_1, a_2, \ldots, a_p\}$ and $B = \{b_1, b_2, \ldots, b_q\}$ and define a p-by-q "adjacency" matrix M for the pair (A, B) with rows indexed by the elements of A and columns by the elements of B as follows.

For each $a_i \in A$ and $b_j \in B$ the entry $m(a_i, b_j)$ in the row of a_i and column of b_j is given by

$$m(a_i, b_j) = \begin{cases} \lambda & \text{if } (a_i, b_j) \in E, \\ -1 & \text{if } (a_i, b_j) \notin E. \end{cases}$$

Let $A' \subseteq A$ and $B' \subseteq B$ with $|A'| = k$, $|B'| = \ell$, where $k \geq \delta|A|$ and $\ell \geq \delta|B|$. Our aim is to show that the density of the pair (A', B') satisfies

$$\rho - \delta < d(A', B') < \rho + \delta.$$

Let E' be the subset of E consisting of all edges of G joining a vertex of A' to a vertex of B'. By reordering we may assume that $A' = \{a_1, a_2, \ldots, a_k\}$ and $B' = \{b_1, b_2, \ldots, b_\ell\}$. Let M' be the k-by-ℓ submatrix of M associated with A' and B'. That is,

$$M' = (m(a_i, b_j))_{1 \le i \le k, 1 \le j \le \ell}.$$

The sum of all of the entries of M' is equal to λ times the number of edges in E' minus the number of nonedges:

$$\sum_{i=1}^{k} \sum_{j=1}^{\ell} m(a_i, b_j) = \lambda |E'| - (k \cdot \ell - |E'|). \tag{11}$$

For $a_i \in A'$ let $\vec{a}_i$ be the corresponding row vector of M and let $\vec{a}'_i$ be the corresponding row vector of M'.

Then by the Cauchy–Schwartz inequality,

$$\frac{1}{\ell} \left(\sum_{i=1}^{k} \sum_{j=1}^{\ell} m(a_i, b_j) \right)^2 \le \sum_{j=1}^{\ell} \left(\sum_{i=1}^{k} m(a_i, b_j) \right)^2 = \left(\sum_{i=1}^{k} \vec{a}'_i \right)^2, \tag{12}$$

where for vectors $\vec{x}$ and $\vec{y}$ the expression $\vec{x}\vec{y}$ means the usual scalar product. Clearly

$$\left(\sum_{i=1}^{k} \vec{a}'_j \right)^2 \le \left(\sum_{i=1}^{k} \vec{a}_i \right)^2, \tag{13}$$

therefore by (11), (12), and (13) we have

$$(\lambda \cdot |E'| - (k \cdot \ell - |E'|))^2 \le \ell \left(\sum_{i=1}^{k} \vec{a}_i \right)^2. \tag{14}$$

In what follows we will compute an upper bound for

$$\left(\sum_{i=1}^{k} \vec{a}_i \right)^2 = \sum_{i=1}^{k} \vec{a}_i^2 + 2 \sum_{1 \le i < j \le k} \vec{a}_i \cdot \vec{a}_j.$$

For each $a_i \in A$ we have

$$\vec{a}_i^2 = \lambda^2 \deg(a_i) + q - \deg(a_i)$$

and hence, since $0 \le \deg(a_i) \le q$,

$$\vec{a}_i^2 \le \max\{q, \lambda^2 q\}. \tag{15}$$

For $a_i \ne a_j \in A$ the product $\vec{a}_i \cdot \vec{a}_j$ is maximized when at each coordinate the entries of $\vec{a}_i$ and $\vec{a}_j$ are equal, so here too we have

$$\vec{a}_i \cdot \vec{a}_j \le \lambda^2 \deg(a_i) + q - \deg(a_i)$$

and hence that

$$\vec{a}_i \cdot \vec{a}_j \le \max\{q, \lambda^2 q\}. \tag{16}$$

In any case we have

$$\vec{a}_i \cdot \vec{a}_j = \lambda^2 \deg(a_i, a_j) - \lambda(\deg(a_i) - \deg(a_i, a_j)) - \lambda(\deg(a_j) - \deg(a_i, a_j)) + (q - \deg(a_i) - \deg(a_j) + \deg(a_i, a_j)).$$

If $\{a_i, a_j\} \in D$, then we have

$$\deg(a_i, a_j) > (\rho - \epsilon)q \quad \text{and} \quad \deg(a_i, a_j) < (\rho + \epsilon)^2 q,$$

thus for such a pair,

$$\vec{a}_i \cdot \vec{a}_j < (\lambda^2 + 2\lambda + 1)(\rho + \epsilon)^2 q - 2(\lambda + 1)(\rho - \epsilon)q + q.$$

By our choice of λ we have that $\lambda + 1 = \frac{1}{\rho}$, so that

$$\vec{a}_i \cdot \vec{a}_j < \frac{(\rho + \epsilon)^2}{\rho^2} q - \frac{2(\rho - \epsilon)}{\rho} q + q = \left(\frac{4\epsilon}{\rho} + \frac{\epsilon^2}{\rho^2}\right) q. \tag{17}$$

Since $\binom{k}{2} - |D| \le \binom{p}{2} - |D| < \frac{5}{2}\epsilon p^2$, we obtain from (15), (16), and (17)

$$\left(\sum_{1 \le i \le k} \vec{a}_i\right)^2 = \left(\sum_{1 \le i \le k} \vec{a}_i^2\right) + 2\left(\sum_{\substack{1 \le i < j \le k \\ \{a_i, a_j\} \notin D}} \vec{a}_i \cdot \vec{a}_j\right) + 2\left(\sum_{\substack{1 \le i < j \le k \\ \{a_i, a_j\} \in D}} \vec{a}_i \cdot \vec{a}_j\right)$$

$$< k \max\{q, \lambda^2 q\} + 5\epsilon p^2 \max\{q, \lambda^2 q\} + 2p^2 q \left(\frac{4\epsilon}{\rho} + \frac{\epsilon^2}{\rho^2}\right).$$

Hence equation (14) becomes

$$(\lambda |E'| - k\ell + |E'|)^2 < k\ell q \max\{1, \lambda^2\} + 5\epsilon \ell p^2 q \max\{1, \lambda^2\} + 2\ell p^2 q \left(\frac{4\epsilon}{\rho} + \frac{\epsilon^2}{\rho^2}\right), \tag{18}$$

and, using the fact that $\lambda = \frac{1-\rho}{\rho}$, we have

$$\left(\frac{1}{\rho}|E'| - k\ell\right)^2 < k\ell q \max\left\{1, \frac{(1-\rho)^2}{\rho^2}\right\} + 5\epsilon \ell p^2 q \max\left\{1, \frac{(1-\rho)^2}{\rho^2}\right\} + 2\ell p^2 q \left(\frac{4\epsilon}{\rho} + \frac{\epsilon^2}{\rho^2}\right), \tag{19}$$

or equivalently

$$\left||E'| - \rho k\ell\right| < \sqrt{k\ell q \max\{\rho^2, (1-\rho)^2\} + 5\epsilon \ell p^2 q \max\{\rho^2, (1-\rho)^2\} + 2\ell p^2 q(4\epsilon\rho + \epsilon^2)}.$$

Thus, since $0 < \rho \le 1$, we have

$$\left||E'| - \rho k\ell\right| < \sqrt{k\ell q + 5\ell\epsilon p^2 q + 2\ell p^2 q(4\epsilon + \epsilon^2)}. \tag{20}$$

Therefore

$$\left|\frac{|E'|}{k\ell} - \rho\right| < \sqrt{\frac{q}{k\ell} + \frac{5\epsilon p^2 q}{k^2\ell} + \frac{2p^2q}{k^2\ell}(4\epsilon + \epsilon^2)},$$

which, since $k \geq \delta p$ and $\ell \geq \delta q$, yields

$$\left|\frac{|E'|}{k\ell} - \rho\right| < \sqrt{\frac{1}{\delta^2 p} + \frac{13\epsilon + 2\epsilon^2}{\delta^3}} \leq \sqrt{\frac{1}{\delta^2 p} + \frac{15\epsilon}{\delta^3}}. \tag{21}$$

Since $p = |A| \geq \frac{2}{\epsilon}$ and $\delta = (16\epsilon)^{1/5}$ we have

$$p \geq \frac{2}{\epsilon}\left(\frac{\epsilon}{2}\right)^{1/5} = \frac{\delta}{\epsilon},$$

and hence that

$$\frac{1}{\delta^2 p} \leq \frac{\epsilon}{\delta^3}.$$

Thus (21) yields

$$|d(A', B') - d(A, B)| = \left|\frac{|E'|}{k\ell} - \rho\right| < \sqrt{\frac{16\epsilon}{\delta^3}} = \delta.$$

It follows that the pair (A, B) is δ-regular.

The existence of the sequential algorithm claimed in Lemma 1.4 is now an easy consequence of Propositions 2.3, 2.4, and 2.5. Let a constant ϵ, a graph G, and a pair (A, B) of subsets of the vertices of G be as described in that lemma. Let F be the set of edges of the bipartite subgraph of G induced by the pair (A, B). For each vertex $x \in A$ we compute the degree of x in B, $\deg_B(x)$, and compute the density $d(A, B) = \rho$. This can be done in time $O(N^2)$. Note that we may assume that $\rho|B| \geq 1$. Otherwise we would consider the bipartite graph with vertex classes A and B and edge set $\bar{F} = A \times B \backslash F$, and use the fact that for any pair (X, Y), $X \subseteq A$, $Y \subseteq B$, the density $\bar{d}(X, Y)$ of this pair with respect to $\bar{F}$ satisfies $\bar{d}(X, Y) = 1 - d(X, Y)$. It follows that (A, B) is ϵ-regular with respect to the edge set F if and only if it is ϵ-regular with respect to $\bar{F}$ and that

$$|\bar{d}(A, B) - \bar{d}(X, Y)| \geq \delta$$

implies that

$$|d(A, B) - d(X, Y)| \geq \delta.$$

Let $\delta = \epsilon^5/16$ and set $Z = \{x \in A | (\rho - \epsilon)|B| < \deg_B(x) < (\rho + \epsilon)|B|\}$.

1. If $|Z| \leq (1 - 2\delta)|A|$, then by Proposition 2.3 the pair (A, B) is not δ-regular, and, as in the proof of that proposition, it is trivial to find a subset $X \subseteq A \backslash Z$ with $|X| \geq \delta|A|$ such that $d(X, B) \leq \rho - \delta$ or $d(X, B) \geq \rho + \delta$.

2. If $|Z| > (1 - 2\delta)|A|$, let M be the $|Z|$-by-$|B|$ matrix with rows indexed by the elements of Z and columns by the elements of B in which the entry $m(x, y)$ in the row of x and column of y is given by

$$m(x, y) = \begin{cases} 1 & \text{if } (x, y) \in F, \\ 0 & \text{otherwise.} \end{cases}$$

The entries of the matrix product $M \times M^t$, where M^t is the transpose of M, are the codegrees of the pairs of vertices in Z. It follows that these codegrees can be computed in $O(M(N))$ time. For each $x \in Z$ let

$$D_x = \{x' \in Z \backslash \{x\} | \deg_B(x, x') < (\rho + \epsilon)^2 |B|\}.$$

If for some $x_0 \in Z$ we have $|D_{x_0}| < (1-3\delta)|A|$, then as in the proof of Proposition 2.4, the set $X = \{x' \in A \backslash \{x\} | \deg_B(x, x') \geq (\rho+\delta)^2 |B|\}$ satisfies $|X| \geq \delta |A|$ and by choosing Y to be any subset of B which contains all the neighbors of x in B and for which $\delta |B| \leq |Y| \leq (\rho+\delta)|B|$, we obtain a pair (X, Y) whose density differs from $d(A, B)$ by at least δ.

3. If, on the other hand, $|Z| > (1-2\delta)|A|$ and $|D_x| \geq (1-3\delta)|A|$ for each $x \in Z$, then since $\epsilon = \delta^5/16$ and $|A| \geq (\frac{2}{\epsilon})^5 = \frac{2}{\delta}$, it follows from Proposition 2.5 that the pair (A, B) is ϵ-regular.

This completes the proof of Lemma 1.4.

Corollary 1.5 now follows by combining the proof of the existence of an ϵ-regular partition of k-tuples as given for Proposition 2.1 with Lemma 1.4. Suppose we are given a positive constant ϵ and a k–partite graph $G = (V, E)$, $V = \cup_{1 \leq i \leq k} V_i$, $|V_i| = N$, $i = 1, 2, \ldots, k$, as in the corollary, where $\delta^{q+1} N \geq 2$ when $\delta = \epsilon^5/16$ and $q = \binom{k}{2}/\epsilon\delta^4$.

1. We let $\mathcal{K}^0 = \{K_1\}$, $K_1 = V_1 \times V_2 \times \cdots \times V_k$, be the first partition and compute the $\binom{k}{2}$ densities of the pairs (V_i, V_j), $1 \leq i < j \leq k$.

2. Given a partition $\mathcal{K}^m$, for every pair $(V_i(K), V_j(K))$ of each cylinder K in that partition we use the algorithm of Lemma 1.4 to verify that this pair is ϵ-regular or to find subsets $W_i \subseteq V_i(K)$, $W_j \subseteq V_j(K)$, $|W_i| \geq \delta |V_i(K)|$, $|W_j| \geq \delta |V_j(K)|$, such that $|d(W_i, W_j) - d(V_i(K), V_j(K))| \geq \delta$.

3. If all but ϵN^k of the k-tuples of $V_1 \times V_2 \times \cdots \times V_k$ are contained in ϵ-regular cylinders, then we halt. $\mathcal{K}^m$ is an ϵ-regular partition with at most 4^m cylinders.

4. If more than ϵN^k k-tuples are contained in cylinders which are not ϵ-regular, then for each cylinder K of $\mathcal{K}^m$ which has not been shown to be ϵ-regular choose one of the pairs (W_i, W_j) for a pair $(V_i(K), V_j(K))$ of that cylinder that was found in Step 2 and use this pair, as in the proof of Proposition 2.1, to replace K by four new cylinders. Call the resulting partition $\mathcal{K}'$.

5. Set $\mathcal{K}^{m+1} = \mathcal{K}'$ and $m = m + 1$ and return to Step 2.

As in the proof of Proposition 2.1, we have that $\text{ind}(\mathcal{K}^m) \geq \epsilon\delta^4$ and hence that this procedure halts after at most $q = \binom{k}{2}/\epsilon\delta^4$ iterations. Since $\delta^{q+1} N \geq 2$ we have $\delta^q N \geq (\frac{2}{\epsilon})^5$ which assures that at each iteration the pairs $(V_i(K), V_j(K))$ are large enough for Step 2 to be carried out using the algorithm of Lemma 1.4.

Observe that by our construction each k-tuple of $V_1 \times V_2 \times \cdots \times V_k$ is contained in at most one ϵ-regular cyclinder. Hence, the number of k-tuples in $K = W_1 \times W_2 \times \cdots \times W_k$ is given by $\prod_{i=1}^k |W_i|$. Thus checking whether all but ϵN^k of the k-tuples of $V_1 \times V_2 \times \cdots \times V_k$ are contained in ϵ-regular cyclinders takes by Proposition 2.1 at most $O(4^{\binom{k}{2}/\epsilon\delta^4} \cdot n)$ sequential time.

Finally, since each cylinder contains $\binom{k}{2}$ pairs and the partition $\mathcal{K}^m$ has at most 4^m such cylinders, the total number of times that the algorithm of Lemma 1.4 is used is at most $\binom{k}{2} 4^{q+1}$. This establishes that the entire procedure can be implemented in the asserted time and completes the proof of Corollary 1.5.

3. Proof of Lemma 1.6. First we prove a result concerning the number of copies of K_k, the complete graph on k vertices, in a k-partite graph with ϵ-regular pairs of classes, from which Lemma 1.6 follows easily.

PROPOSITION 3.1. *Let $G = (V, E)$, $V = \cup_{1 \leq i \leq k} V_i$, be a k-partite graph, $k \geq 3$, and for each i and j, $1 \leq i < j \leq k$, let $d_{i,j}$ be the density of the pair (V_i, V_j). Suppose that δ is*

a positive constant and that for each i and j, $1 \le i < j \le k$, the pair (V_i, V_j) is ϵ-regular, where $\epsilon = \delta^2/k^3$. Then the number $f_k(G)$ of subgraphs of G which are isomorphic to K_k, $k \ge 3$, satisfies the following:

$$\left| f_k(G) - \prod_{1 \le i < j \le k} d_{i,j} \cdot \prod_{1 \le i \le k} |V_i| \right| \le \delta \prod_{1 \le i \le k} |V_i|. \tag{22}$$

Proof. If $\delta \ge 1$ there is nothing to prove, so we assume $\delta < 1$. Note also that (22) is satisfied if $d_{i,j} < \delta$ for some i and j since in this case $f_k(G) < \delta \prod_{1 \le i \le k} |V_i|$, so we also assume that $\delta_{i,j} \ge \delta$ in all cases.

For integers j and ℓ, $1 \le j < \ell \le k$, and any sequence $x_1, x_2, \ldots, x_k$, $x_i \in V_i$, $i = 1, 2, \ldots k$, we will let $N_\ell(x_1, x_2, \ldots, x_j)$ denote the set of common neighbors of $x_1, x_2, \ldots,$ and x_j in V_ℓ, and we call the sequence $x_1, x_2, \ldots, x_k$ "good" if it satisfies

$$|V_\ell| \prod_{1 \le i \le j} (d_{i,\ell} - \epsilon) \le |N_\ell(x_1, x_2, \ldots, x_j)| \le |V_\ell| \prod_{1 \le i \le j} (d_{i,\ell} + \epsilon), \tag{23}$$

for each j, $1 \le j \le k-1$, and each ℓ, $j < \ell \le k$.

First we seek upper and lower bounds for $f_k(G)$ when the following condition holds:

$$\prod_{1 \le i \le j} (d_{i,\ell} - \epsilon) > \epsilon \quad \text{for each } j \text{ and } \ell, \quad 1 \le j < \ell \le k. \tag{24}$$

Assuming (24) does hold we wish to find an upper bound for the number of copies of K_k in G which are spanned by good sequences. Trivially there are at most $|V_1|$ choices for the first vertex in such a sequence. Suppose $x_1, x_2, \ldots, x_{j-1}$ are the first $j-1$ vertices of some good sequence which spans a copy of K_k in G. Then the next vertex of this sequence, x_j, must be in $N_j(x_1, x_2, \ldots, x_{j-1})$. By the definition of "good sequence" there are at most $|V_j| \prod_{1 \le i \le j-1} (d_{i,j} + \epsilon)$ choices for x_j for each such sequence $x_1, x_2, \ldots, x_{j-1}$. It follows that the number of copies of K_k spanned by good sequences is at most

$$\prod_{1 \le i \le k} |V_i| \prod_{1 \le i < j \le k} (d_{i,j} + \epsilon). \tag{25}$$

Now consider the number of copies of K_k spanned by sequences which are not good (still asuming that (24) holds). For each ℓ, $2 \le \ell \le k$, the ϵ-regularity of the pair (V_1, V_ℓ) implies by Proposition 2.3 that there are more than $(1 - 2\epsilon)|V_1|$ vertices in V_1 for which we have

$$|V_\ell|(d_{1,\ell} - \epsilon) \le |N_\ell(x_1)| \le |V_\ell|(d_{1,\ell} + \epsilon). \tag{26}$$

It follows that there are more than $(1 - 2(k-1)\epsilon)|V_1|$ vertices in V_1 for which (26) holds for each ℓ, $2 \le \ell \le k$. The number of copies of K_k spanned by sequences where (23) does not hold for $j = 1$ and at least one value of ℓ, $2 \le \ell \le k$, is therefore less than $2(k-1)\epsilon \prod_{1 \le i \le k} |V_i|$. Suppose the sequence $x_1, x_2, \ldots, x_k$ spans a copy of K_k and that there exists an integer s, $2 \le s \le k-1$, such that (23) holds for each j, $1 \le j \le s-1$, and each ℓ, $j < \ell \le k$, but that (23) does not hold for $x_1, x_2, \ldots, x_s$ and some particular value of ℓ, say $\ell = t$ $(t > s)$. Then we have $|N_t(x_1, x_2, \ldots, x_{s-1})| \ge |V_t| \prod_{1 \le i \le s-1} (d_{i,t} - \epsilon)$ and, given that (24) holds, $|N_t(x_1, x_2, \ldots, x_{s-1})| \ge \epsilon |V_t|$. Hence as in the proof of Proposition 2.2 (resp., 2.3) there are fewer than $2\epsilon|V_s|$ vertices in V_s which have more than $(d_{s,t} + \epsilon)|N_t(x_1, x_2, \ldots, x_{s-1})|$ or fewer than $(d_{s,t} - \epsilon)|N_t(x_1, x_2, \ldots, x_{s-1})|$ neighbors in $N_t(x_1, x_2, \ldots, x_{s-1})$. Thus there are fewer than $2(k-s)\epsilon|V_s|$ vertices x_s such that (23) holds for each j, $1 \le j \le s-1$, but

such that it does not hold for $x_1, x_2, \ldots, x_s$ and at least one value of ℓ greater than s. It follows that fewer than $2(k-s)\epsilon \prod_{1\le i\le k} |V_i|$ copies of K_k are spanned by sequences of this type.

We now have that when (24) holds the number of copies of K_k spanned by sequences which are not good is less than

$$\prod_{1\le i\le k} |V_i| \sum_{1\le s\le k-1} 2\epsilon(k-s) = k(k-1)\epsilon \prod_{1\le i\le k} |V_i|. \tag{27}$$

Combining (25) and (27) shows that in this case

$$f_k(G) < \left[k(k-1)\epsilon + \prod_{1\le i<j\le k} (d_{i,j}+\epsilon)\right] \prod_{1\le i\le k} |V_i|. \tag{28}$$

Since $d_{i,j} \ge \delta = k^3\epsilon/\delta$ for each i and j, $1 \le i < j \le k$, we have

$$\prod_{1\le i<j\le k} (d_{i,j}+\epsilon) \le \left(1+\frac{\delta}{k^3}\right)^{\binom{k}{2}} \prod_{1\le i<j\le k} d_{i,j}. \tag{29}$$

Now since

$$\left(1+\frac{\delta}{k^3}\right)^{\binom{k}{2}} < \exp\left[\binom{k}{2}\frac{\delta}{k^3}\right],$$

and $\exp(x) < 1+2x$ for $0 < x < 1$, we have from (29) that

$$\prod_{1\le i<j\le k} (d_{i,j}+\epsilon) < \left(1+\frac{k(k-1)}{k^3}\,\delta\right) \prod_{1\le i<j\le k} d_{i,j} \le \prod_{1\le i<j\le k} d_{i,j} + \frac{k(k-1)}{k^3}\,\delta. \tag{30}$$

It follows that the right-hand side of (28) differs from $\prod_{1\le i<j\le k} d_{i,j} \cdot \prod_{1\le i\le k} |V_i|$ by less than

$$\frac{k(k-1)}{k^3}\,(\delta^2+\delta) \prod_{1\le i\le k} |V_i|, \tag{31}$$

and this quantity is less than $\delta \prod_{1\le i\le k} |V_i|$, since $1+\delta \le k^2/k - 1$ when $k \ge 3$.

Next we wish to find a lower bound for $f_k(G)$ when (24) holds. We will count only those copies of K_k which are spanned by sequences $x_1, x_2, \ldots, x_k$ such that for each j, $1 \le j \le k-1$, and each ℓ, $j < \ell \le k$, we have

$$|N_\ell(x_1, x_2, \ldots, x_j)| \ge |V_\ell| \prod_{1\le i\le j} (d_{i,\ell} - \epsilon). \tag{32}$$

There are at least $(1-(k-1)\epsilon)|V_1|$ choices of a vertex x_1 for which (32) holds for $j = 1$ and each ℓ, $2 \le \ell \le k$, by Proposition 2.2. Suppose $x_1, x_2, \ldots, x_s$, $s \ge 2$, is a sequence of vertices in $V_1 \times V_2 \times \cdots \times V_s$ which spans a copy of K_s in G and for which (32) holds for each j, $1 \le j \le s-1$, and each ℓ, $s \le \ell \le k$. Then for each ℓ, $\ell \ge s$, we have $|N_\ell(x_1, x_2, \ldots, x_{s-1})| \ge |V_\ell| \prod_{1\le i\le s-1} (d_{i,\ell} - \epsilon)$ which, given that (24) holds, implies that $|N_\ell(x_1, x_2, \ldots, x_{s-1})| \ge \epsilon|V_\ell|$. Thus for each such ℓ, by ϵ-regularity of the pair (V_s, V_ℓ), there are at most $\epsilon|V_s|$ vertices in V_s having fewer than $(d_{s,\ell}-\epsilon)|N_\ell(x_1, x_2, \ldots, x_{s-1})|$ neighbors in $N_\ell(x_1, x_2, \ldots, x_{s-1})$. That is, there are at least $(1-\epsilon)|V_s|$ vertices x_s in V_s such that $|N_\ell(x_1, x_2, \ldots, x_s)| \ge |V_\ell| \prod_{1\le i\le s} (d_{i,\ell} - \epsilon)$. It follows that there are at least

$$\left[\prod_{1\le i\le s-1} (d_{i,s} - \epsilon) - (k-s)\epsilon\right] |V_s| \tag{33}$$

choices of the vertex x_s in V_s such that $x_1, x_2, \ldots, x_s$ span a copy of K_s and for which (32) holds for each j, $1 \le j \le s$, and each ℓ, $s \le \ell \le k$. Since $d_{i,j} \le 1$ and $d_{i,j} - \epsilon \ge 0$ for each i and j, $1 \le i < j \le k$, it can easily be shown by induction on s that

$$\prod_{1 \le i \le s-1} (d_{i,s} - \epsilon) \ge \prod_{1 \le i \le s-1} d_{i,s} - (s-1)\epsilon.$$

Hence the quantity in (33) is at least

$$\left[\prod_{1 \le i \le s-1} d_{i,s} - (k-1)\epsilon \right] |V_s|,$$

and when (24) holds we have

$$f_k(G) \ge \prod_{1 \le i \le k} |V_i| \prod_{1 \le s \le k} \left[\prod_{1 \le i \le s-1} d_{i,s} - (k-1)\epsilon \right]. \tag{34}$$

If for each s, $1 \le s \le k-1$, we have $\prod_{1 \le i \le s-1} d_{i,s} \ge (k-1)\epsilon$, then

$$\prod_{1 \le s \le k} \left[\prod_{1 \le i \le s-1} d_{i,s} - (k-1)\epsilon \right] \ge \prod_{1 \le i < j \le k} d_{i,j} - k(k-1)\epsilon.$$

In this case, since $k(k-1)\epsilon = (k(k-1)/k^3)\,\delta^2 < \delta$, the upper bound of (22) follows from (34). If, on the other hand, $\prod_{1 \le i \le s-1} d_{i,s} < (k-1)\epsilon$ for some value of s, then we have $\prod_{1 \le i < j \le k} d_{i,j} < ((k-1)/k^3)\,\delta^2 < \delta$ and in this case the lower bound in (22) is trivially satisfied.

Finally we must establish upper and lower bounds for $f_k(G)$ when (24) does not hold. Assume then that t is the smallest integer, $1 \le t \le k-1$, for which we have $\prod_{1 \le i \le t} (d_{i,\ell} - \epsilon) < \epsilon$ for some value of ℓ, $t < \ell \le k$, and that r is the smallest integer, $r > t$, such that $\prod_{1 \le i \le t} (d_{i,r} - \epsilon) < \epsilon$.

The number of copies of K_k in G which are spanned by good sequences is again bounded from above by the quantity in (25). Suppose that $x_1, x_2, \ldots, x_k$ is a sequence which spans a copy of K_k but which is not good. Let s be the smallest integer such that (23) holds for each j, $1 \le j \le s-1$, and each ℓ, $j < \ell \le k$, but that (23) does not hold for $x_1, x_2, \ldots, x_s$ and at least one value of ℓ, $\ell \ge s$. If $s \le t$ for t as defined above, then for each $\ell \ge s$ we have $|N_\ell(x_1, x_2, \ldots, x_{s-1})| \ge |V_\ell| \prod_{1 \le i \le s-1} (d_{i,\ell} - \epsilon)$ by (23) and therefore that $|N_\ell(x_1, x_2, \ldots, x_{s-1})| \ge \epsilon |V_\ell|$. In this case it follows as before that there are at most $2(k-s)\epsilon \prod_{1 \le i \le k} |V_i|$ copies of K_k spanned by such sequences. If $s > t$, then (23) holds for each j, $j \le t$, and each ℓ, $j < \ell \le k$. Since $x_1, x_2, \ldots, x_k$ spans a copy of K_k, x_r must be in $N_r(x_1, x_2, \ldots, x_t)$. By (23) we have $|N_r(x_1, x_2, \ldots, x_t)| \le \prod_{1 \le i \le t} (d_{i,r} + \epsilon)$ (while by the definition of r and t we have $\prod_{1 \le i \le t} (d_{i,r} - \epsilon) < \epsilon$). There are at most $\prod_{1 \le i \le k} |V_i| \cdot \prod_{1 \le i \le t} (d_{i,r} + \epsilon)$ copies of K_k spanned by sequences of this type. Combining these results we have

$$f_k(G) \le \left[\prod_{1 \le i < j \le k} (d_{i,j} + \epsilon) + \sum_{1 \le j \le t} 2(k-j)\epsilon + \prod_{1 \le i \le t} (d_{i,r} + \epsilon) \right] \prod_{1 \le i \le k} |V_i|. \tag{35}$$

Since we have $d_{i,j} \ge \delta = k^3\epsilon/\delta$ in all cases we obtain

$$\frac{d_{i,j} + \epsilon}{d_{i,j} - \epsilon} = 1 + \frac{2\epsilon}{d_{i,j} - \epsilon} \le 1 + 2\left(\frac{k^3}{\delta} - 1\right)^{-1}$$

for each i and j, $1 \le i < j \le k$, and hence that

$$\prod_{1\le i<j\le k}(d_{i,j}+\epsilon) \le \left(1+2\left(\frac{k^3}{\delta}-1\right)^{-1}\right)^{\binom{k}{2}} \prod_{1\le i<j\le k}(d_{i,j}-\epsilon), \tag{36}$$

where

$$\prod_{1\le i<j\le k}(d_{i,j}-\epsilon) \le \prod_{1\le i\le t}(d_{i,r}-\epsilon) < \epsilon. \tag{37}$$

Reasoning as for (30) and using the fact that

$$k(k-1)\left(\frac{k^3}{\delta}-1\right)^{-1} \le k^2\left(\frac{k^3}{\delta}\right)^{-1} = \frac{\delta}{k} < \frac{1}{k},$$

we obtain

$$\left(1+2\left(\frac{k^3}{\delta}-1\right)^{-1}\right)^{t} < \left(1+2\left(\frac{k^3}{\delta}-1\right)^{-1}\right)^{\binom{k}{2}} < 1+\frac{2}{k}. \tag{38}$$

Finally since $t \le k-1$, we have $\sum_{1\le j\le t} 2(k-j) \le k(k-1)$. Combining (35), (36), (37), and (38) yields

$$f_k(G) \le \left[2\left(1+\frac{2}{k}\right)\epsilon + k(k-1)\epsilon\right] \prod_{1\le i\le k}|V_i|, \tag{39}$$

which leads to

$$f_k(G) \le \left[\frac{4}{k^3}+\frac{k(k-1)}{k^3}\right]\delta^2 \prod_{1\le i\le k}|V_i| < \left(\frac{2}{k}\right)\delta^2 \prod_{1\le i\le k}|V_i| < \delta \prod_{1\le i\le k}|V_i|.$$

Thus the upper bound in (22) holds in this case.

Given that $\prod_{1\le i\le t}(d_{i,r}-\epsilon) < \epsilon$ for some r and t, we have $\prod_{1\le i<j\le k}(d_{i,j}-\epsilon) < \epsilon$, so, again using $d_{i,j} \ge \delta$ for all i and j, $1 \le i < j \le k$, we obtain

$$\prod_{1\le i<j\le k} d_{i,j} = \prod_{1\le i<j\le k} \frac{d_{i,j}}{d_{i,j}-\epsilon} \prod_{1\le i<j\le k}(d_{i,j}-\epsilon) \le \left(1+\left(\frac{k^3}{\delta}-1\right)^{-1}\right)^{\binom{k}{2}} \epsilon.$$

Thus, as in (30) and (38), we have

$$\prod_{1\le i<j\le k} d_{i,j} < 1+\frac{1}{k}\epsilon < 2\,\frac{\delta^2}{k^3} < \delta. \tag{40}$$

The lower bound in (22) is trivially satisfied in this case, which completes the proof of the proposition.

To establish Lemma 1.6 we must consider an estimate for $f_<(H, G)$, the number of induced subgraphs of G which are partite isomorphic to the k-vertex graph H. To do so we simply make use of the graph $\hat{G}$ which is obtained from G by replacing the bipartite subgraph induced by V_i and V_j by its bipartite complement whenever (w_i, w_j) is not an edge of H. Then we have $f_<(H, G) = f_k(\hat{G})$. Since ϵ-regularity is preserved with respect to taking complements and as the density of the pair (V_i, V_j) in $\hat{G}$ is $\hat{d}_{i,j}$, Lemma 1.6 follows immediately from Proposition 3.1.

4. The proof of Theorem 1.1. Let c be a constant $0 < c \leq 1$, n be an integer for which $\log\log n \leq \sqrt{c \log n}$, and k be an integer satisfying $3 \leq k \leq \sqrt{c \log n}$. Let $G = (V, E)$ be a graph with an n-element vertex set $V = \{v_1, v_2, \ldots, v_n\}$ ordered by $v_1 < v_2 < \cdots < v_n$ and suppose we are given a list $H_1, H_2, \ldots, H_t$, $t = 2^{\binom{k}{2}}$, of all labeled graphs on the vertex set $W = \{w_1, w_2, \ldots, w_k\}$ ordered by $w_1 < w_2 < \cdots < w_k$.

We begin by partitioning V into $m = \lceil \frac{k^2}{\epsilon} \rceil$ parts $V_1, V_2, \ldots, V_m$ each of size $N = n/m$ (where w.l.o.g. m divides n) with

$$V_j = \{v_{(j-1)N+1}, v_{(j-1)N+2}, \ldots, v_{jN}\}$$

for $j = 1, 2, \ldots, m$.

For each of the $\binom{m}{k}$ choices of k sets $V_{i_1}, V_{i_2}, \ldots, V_{i_k}$, with $1 \leq i_1 < i_2 < \cdots < i_k \leq m$ we use the algorithm of Corollary 1.5 to obtain an ϵ-regular partition of the k-partite subgraph of G induced by $\cup_{1 \leq j \leq k} V_{i_j}$, with ϵ as in the statement of the theorem. For each ϵ-regular cylinder of each such partition and for each labeled graph H_i, $1 \leq i \leq t$, we use the inequality of Lemma 1.6 to obtain an estimate of the number of induced subgraphs of G which have as vertices a k-tuple in that cylinder and which are partite isomorphic to H_i. For each i, $1 \leq i \leq t$, we add these estimates to obtain the entry $\bar{h}_i$ of $\bar{\sigma}_k$. Thus the remainder of the proof of Theorem 1.1 consists of establishing the following:

(i) The conditions of Corollary 1.5 are satisfied whenever the algorithm in that statement is employed,

(ii) For each i, $1 \leq i \leq t$, the error $|h_i - \bar{h}_i|$ is at most $2k(k\epsilon)^{1/2}$, and

(iii) This procedure can be implemented in $O(2^{\binom{k}{2}} n^{2c} M(n))$ sequential time as asserted by the theorem.

For (i) it suffices to show that $\delta^{q+1} N \geq 2$, where $\delta = \epsilon^5/16$, $q = \binom{k}{2}/\epsilon\delta^4$, and $N = \frac{n}{m}$ as above. First note that by the definition of ϵ, namely $\epsilon = (2^{16} k^2 \log\log n / c \log n)^{1/21}$, and the fact that $k \geq 3$ we have

$$\log \epsilon \geq \frac{1}{21}(16 + \log k^2 - \log\log n) > \frac{1}{21}(19 - \log\log n). \tag{41}$$

It follows that

$$\log(1/\delta) \leq 4 - \frac{5}{21}(19 - \log\log n) < \frac{5}{21}\log\log n. \tag{42}$$

From the definition of q we have

$$q = \binom{k}{2}\frac{2^{16}}{\epsilon^{21}} = \frac{\binom{k}{2} c \log n}{k^2 \log\log n} < \frac{c \log n}{2 \log\log n}. \tag{43}$$

Since $k \geq 3$ and $\log\log n \leq \sqrt{c \log n}$ we have $2^{16} \log\log n < k^{19} c \log n$, from which it follows that

$$\epsilon < k < \frac{1}{2} k^2. \tag{44}$$

This yields

$$m = \left\lceil \frac{k^2}{\epsilon} \right\rceil < \frac{k^2 + \epsilon}{\epsilon} < \frac{2k^2}{\epsilon}, \tag{45}$$

and therefore

$$\log N = \log n - \log m > \log n - \log k^2 + \log \epsilon - 1. \tag{46}$$

Combining (41) and (46) we have

$$\log N > \log n - \log k^2 + \frac{1}{21}\log k^2 - \frac{1}{21}\log\log n - 1 \geq \log n - \log\log n - 1. \tag{47}$$

From (42) and (43) we obtain

$$\log(\delta^{q+1}N) > \log N - \frac{5}{42}\log n - \frac{5}{21}\log\log n \tag{48}$$

and using (47),

$$\log(\delta^{q+1}N) > \frac{37}{42}\log n - \frac{26}{21}\log\log n - 1. \tag{49}$$

For (i) it suffices to show that $\log(\delta^{q+1}N) \geq 1$, or by (49), that $37\log n - 52\log\log n \geq 84$, which follows from the fact that n satisfies $\log\log n \leq \sqrt{\log n}$.

For each i, $1 \leq i \leq t$, the error term $|h_i - \bar{h}_i|$ in (ii) is at most the sum of the following three quantities:

a) The number of k-element subsets of the vertex set V which meet at least one of the classes of V_{i_j} in more than one point,

b) The number of k-tuples in $V_{i_1} \times V_{i_2} \times \cdots \times V_{i_k}$ for each of the $\binom{m}{k}$ choices of the classes $V_{i_1}, V_{i_2}, \ldots, V_{i_k}$ which are not in ϵ-regular cylinders of the partition obtained for the k-partite subgraph of G induced by $\cup_{1\leq j\leq k} V_{i_j}$, and

c) The total error resulting from using the estimate of Lemma 1.6 of the number of induced subgraphs of G which are partite isomorphic to H_i for each ϵ-regular cylinder of each of the $\binom{m}{k}$ partitions.

To establish (ii) it suffices to show that the sum of the quantities in (a), (b), and (c) is at most $2k(k\epsilon)^{1/2}$.

The number of choices of a pair of vertices of G which are in the same class V_{i_j} is $m\binom{N}{2}$. Since $m = \lceil \frac{k^2}{\epsilon} \rceil$ we have

$$m\binom{N}{2} < \frac{1}{m}\binom{n}{2} \leq \frac{\epsilon}{k^2}\binom{n}{2}. \tag{50}$$

Since each such pair is in $\binom{n-2}{k-2}$ k-tuples, the quantity in (a) is at most

$$\frac{\epsilon}{k^2}\binom{n}{2}\binom{n-2}{k-2} = \frac{\epsilon}{k^2}\binom{k}{2}\binom{n}{k} < \frac{\epsilon}{2}\binom{n}{k}. \tag{51}$$

The number of k-tuples of vertices with at most one point in each class V_{i_j}, but not in an ϵ-regular cylinder of one of the $\binom{m}{k}$ partitions, is at most $\binom{m}{k}\epsilon N^k$ by Corollary 1.5, so the contribution of (b) to the error satisfies

$$\binom{m}{k}\epsilon N^k = \binom{m}{k}\epsilon\left(\frac{n}{m}\right)^k < \epsilon\binom{n}{k}. \tag{52}$$

By Lemma 1.6 the error resulting from using the inequality of that statement for one ϵ-regular cylinder, $W_{i_1} \times W_{i_2} \times \cdots \times W_{i_k}$, of the partition $\mathcal{K}$ of $V_{i_1} \times V_{i_2} \times \cdots \times V_{i_k}$ is at most $\delta \prod_{1\leq j\leq k} |W_{i_j}|$, where $\delta = k(k\epsilon)^{1/2}$. Since

$$\sum \prod_{1\leq j\leq k} |W_{i_j}| = \prod_{1\leq j\leq k} |V_{i_j}|,$$

where the sum is over all cylinders of $\mathcal{K}$, the contribution from a single partition is $k(k\epsilon)^{1/2}N^k$. The quantity in (c) therefore satisfies

$$\binom{m}{k}k(k\epsilon)^{1/2}N^k < k(k\epsilon)^{1/2}\binom{n}{k}. \tag{53}$$

Combining (51), (52), and (53) we have for each i, $1 \le i \le t$,

$$|h_i - \bar{h}_i| < \left[\frac{3}{2}\,\epsilon + k(k\epsilon)^{1/2}\right]\binom{n}{k}. \tag{54}$$

It follows from (44) that $\epsilon < k^3/6$ and hence that $\frac{3}{2}\sqrt{\epsilon} < k^{3/2}$, which together with (54) establishes the claim in (ii).

Since our algorithm consists of using the procedure described in Corollary 1.5 once for each labeled graph H_i, $1 \le i \le 2^{\binom{k}{2}}$, and each choice of k classes $V_{i_1}, V_{i_2}, \ldots, V_{i_k}$, it can be implemented in $O(2^{\binom{k}{2}}\binom{m}{k}\binom{k}{2}4^{q+1}M(N))$ sequential time. Thus, since $N \le n$, to verify (iii) it suffices to show that

$$\binom{m}{k}\binom{k}{2}4^{q+1} = O(n^{\frac{11}{6}c}). \tag{55}$$

By the definition of m and using (45) we have

$$\binom{m}{k} = \binom{\lceil k^2/\epsilon \rceil}{k} < \binom{2k^2/\epsilon}{k} \le \left(\frac{2ke}{\epsilon}\right)^k. \tag{56}$$

It follows that

$$\log\binom{m}{k} < \frac{5}{2}\,k + k\log k - k\log\epsilon, \tag{57}$$

and, by (41), that

$$\log\binom{m}{k} < \frac{5}{2}\,k + k\log k - \frac{k}{21}\,(16 + \log k^2 - \log\log n). \tag{58}$$

Thus we have

$$\log\binom{m}{k} < 2k + \frac{k}{2}\,\log\log n, \tag{59}$$

since $\log k \le \frac{1}{2}\log\log n$. Now, making use of $\sqrt{c\log n} \ge 3, k$, and $\log\log n$, we obtain

$$\log\binom{m}{k} < \frac{7}{6}\,c\log n, \tag{60}$$

which yields

$$\binom{m}{k} < n^{7c/6}. \tag{61}$$

Again using $3 \le \sqrt{c\log n}$ and $\log\log n \le \sqrt{c\log n}$ we have

$$\log\log n \le \sqrt{c\log n} < \frac{c}{3}\log n. \tag{62}$$

Since $k \leq \sqrt{c \log n}$, it follows that

$$\binom{k}{2} < \frac{c}{2} \log n \leq \frac{c}{2} n^{c/3}. \tag{63}$$

Finally, from (43) we have

$$4^{q+1} \leq 4 \cdot 2^{(c \log n)/(\log \log n)}, \tag{64}$$

which, since $\log \log n \geq 3$, yields

$$4^{q+1} \leq 4 \cdot 2^{(c/3) \log n} = 4n^{c/3}. \tag{65}$$

Combining (61), (63), and (65), we have that (55) is satisfied, which establishes (iii) and hence completes the proof of the theorem.

Notice that the computation above indeed shows that

$$\binom{m}{k}\binom{k}{2}4^{q+1} = O((\log n)^{\frac{7}{6}k+1} \cdot n^{\frac{c}{\log \log n}}) \tag{66}$$

for our choice of the parameters m, k, q, once n is large. To see this, observe that for $\log \log n \geq 3$ we have $2k + \frac{k}{2} \log \log n \leq \frac{7}{6} k \log \log n$ and thus (59) implies $\log \binom{m}{k} < \frac{7}{6} k \log \log n$. This yields a total running time of

$$O\left(2^{\binom{k}{2}} n^{o(1)} M(n)\right) = O\left(n^{\frac{c}{2}+o(1)} M(n)\right).$$

As noted in §1, Corollary 1.2 is an immediate consequence of Theorem 1.1. To verify Corollary 1.3 it must be shown that for $c = \frac{1}{2}$ and $3 \leq k \leq \sqrt{\frac{1}{100} \log \log n}$, we have $2k(k\epsilon)^{1/2} \leq 1/4^{\binom{k}{2}}$. Taking logarithms, this is equivalent to checking that

$$2 + 3 \log k + \log \epsilon \leq 2k - 2k^2. \tag{67}$$

By the definition of ϵ, with $c = \frac{1}{2}$, it is enough to show that

$$\log \log n - 42k^2 + 42k \geq 59 + 65 \log k + \log \log \log n, \tag{68}$$

or, since $k \geq 3$, that

$$\log \log n - 42k^2 \geq 65 \log k + \log \log \log n, \tag{69}$$

which is easily verified when $3 \leq k \leq \sqrt{\frac{1}{100} \log \log n}$.

Concerning the running time, observe that by (66) our choice of the parameters yields

$$\begin{aligned} O\left(2^{\binom{k}{2}}\binom{m}{k}\binom{k}{2}4^{q+1} M(n)\right) &= O\left((\log n)^{1+\frac{7}{6}\sqrt{\frac{1}{100} \log \log n}} n^{\frac{1}{2 \log \log n}} M(n)\right) \\ &= O\left(n^{\frac{1}{\log \log n}} M(n)\right). \end{aligned}$$

5. Remarks and further results. One could combine the algorithm of Lemma 1.4 with the proof of the regularity lemma of Szemerédi [Sz], in much the same way as we obtained Corollary 1.5, to obtain an efficient algorithm which would yield an ϵ-regular partition of

the vertex set of a given graph in the sense of Szemerédi's result. Indeed, a related paper by Alon, Yuster, and the present authors [ADLRY] includes a result (Corollary 3.3) which is essentially an alternate version of Lemma 1.4 and which is used in exactly this way to produce such an algorithm. The resulting constructive version of the regularity lemma (Theorem 1.3 of [ADLRY]) leads to efficient sequential and parallel algorithms for many applications involving topics such as graph colorings and graph decompositions.

We note, however, that in spite of these results it is also shown in [ADLRY] that the problem of deciding whether a given partition of the vertices of an input graph does satisfy the properties guaranteed by Szemerédi's regularity lemma is coNP-complete.

The algorithm for finding an ϵ-regular partition in the sense of the regularity lemma could itself be combined with Lemma 1.6 to obtain an algorithm for computing an approximation to the number of subgraphs of a given graph G which are isomorphic to a particular smaller graph H. Unfortunately, the upper bound on the number of classes in the ϵ-regular partition assured by the "original" regularity lemma is extremely large. Due to the way in which this upper bound depends on ϵ, this approach would only achieve the accuracy of Corollary 1.5 for an input graph G on n vertices when H is a graph of k vertices with $k \leq \log^{(p)} n$, where $\log^{(p)}$ denotes the p-fold iterated logarithm and p in this case is a polynomial (of degree about 20) in $\frac{1}{\epsilon}$. This is the reason for our use of a partition of the set of k-tuples of vertices into cylinders as described in Proposition 2.1.

The regularity result given in Proposition 2.1 is nearly optimal in a certain sense. Although the bound on the number of cylinders in that statment is still exponential in $\frac{1}{\epsilon}$, that this cannot be avoided entirely follows from the existence of a bipartite graph $G = (U \cup V, E)$ in which every "large" pair (X, Y), $X \subseteq U$, $Y \subseteq V$, is either ϵ-irregular or has density satisfying $d(X, Y) \leq \epsilon$. The following results make this statement precise.

THEOREM 5.1. *Given a positive constant ϵ and positive integer n, there exists a bipartite graph $G = (U \cup V, E)$ with $|V| = |U| = n$ and density $d(U, V) = (1 - \sqrt{\epsilon})^{1/2\sqrt{\epsilon}} \sim 1/\sqrt{e}$ such that for any pair (X, Y), $X \subseteq U$, $Y \subseteq V$ for which*

i) $|X||Y| > 4n^2 2^{-1/\sqrt{\epsilon}}$, *and*

ii) $d(X, Y) > \epsilon$,

we have that (X, Y) is ϵ-irregular.

Proof. Let U and V be sets of size n. Set $t = 1/\sqrt{\epsilon}$ and $r = \frac{1}{2}\sqrt{\epsilon}$ (where w.l.o.g. r and t are integers) and partition each of U and V into pairwise disjoint subsets of size n/t^r. Choose a bijection between the t^r subsets of this partition of U and the collection of all sequences of length r whose entries are integers between 1 and t, and a bijection between the t^r subsets of V and this same collection of sequences. If x is in the subset of U which corresponds to the sequence $(u_1, u_2, \ldots, u_r)$, let $x(j) = u_j$ for each j, $j = 1, 2, \ldots, r$. Similarly, for y in V let $y(j)$ denote the jth term of the sequence which corresponds to the subset of V containing y. Define a bipartite graph G on $U \cup V$ as follows: the pair $\{u, v\}$, $u \in U$ and $v \in V$, is an edge of G if and only if $x(j) \neq y(j)$ for each j, $j = 1, 2, \ldots, r$.

Note that each point $u \in U$ is joined to

$$(t-1)^r \frac{n}{t^r} = \left(1 - \frac{1}{t}\right)^r n = \left(1 - \sqrt{\epsilon}\right)^{(1/2)\sqrt{\epsilon}} n$$

vertices of V, from which it follows that

$$d(U, V) = \left(1 - \sqrt{\epsilon}\right)^{(1/2)\sqrt{\epsilon}}.$$

Now suppose that (X, Y) is an ϵ-regular pair with $X \subseteq U$, $Y \subseteq V$, and that this pair has density greater than ϵ.

For each j, $j = 1, 2, \ldots, r$, and each i, $i = 1, 2, \ldots, t$, set

$$X^i(j) = \{x \in X | x(j) = i\},$$

and

$$Y^i(j) = \{y \in Y | y(j) = i\}.$$

Clearly, for each j, $j = 1, 2, \ldots, r$, we have

$$X = \bigcup_{1 \le i \le t} X^i(j) \quad \text{and} \quad Y = \bigcup_{1 \le i \le t} Y^i(j).$$

Now set

$$X(j) = \bigcup_{1 \le i \le t} \{X^i(j) | |X^i(j)| < \epsilon |X|\}$$

and

$$Y(j) = \bigcup_{1 \le i \le t} \{Y^i(j) | |Y^i(j)| < \epsilon |Y|\}$$

for each j, $j = 1, 2, \ldots, r$.

It follows immediately that

$$|X(j)| < t\epsilon |X| = \sqrt{\epsilon}|X| \quad \text{and} \quad |Y(j)| < t\epsilon |Y| = \sqrt{\epsilon}|Y|. \tag{70}$$

Finally, set

$$X' = X - \bigcup_{1 \le j \le r} X(j) \quad \text{and} \quad Y' = Y - \bigcup_{1 \le j \le r} Y(j).$$

From (70) we have that

$$|X'| > (1 - r\sqrt{\epsilon})|X| = \frac{1}{2}|X| \quad \text{and} \quad |Y'| > (1 - r\sqrt{\epsilon})|Y| = \frac{1}{2}|Y|. \tag{71}$$

Notice that by the definition of G no edge joins any vertex of $X^i(j)$ to any vertex of $Y^i(j)$. It follows that the density of such a pair is zero and, since (X, Y) is an ϵ-regular pair with $d(X, Y) > \epsilon$, we have

$$|X^i(j)| < \epsilon |X| \quad \text{or} \quad |Y^i(j)| < \epsilon |Y| \tag{72}$$

for each j, $j = 1, 2, \ldots, r$ and each i, $i = 1, 2, \ldots, t$.

If $x \in X'$ and $y \in Y'$, then by definition we have both $|X^{x(j)}| \ge \epsilon |X|$ and $|Y^{y(j)}| \ge \epsilon |Y|$. It follows from (72) that in this case we have $x(j) \ne y(j)$ for each j, $j = 1, 2, \ldots, r$. Thus for each j there exists a partition $\mathcal{X}_j \cup \mathcal{Y}_j = \{1, 2, \ldots, t\}$ with the property that if $x \in X'$, then $x(j) \in \mathcal{X}_j$, and if $y \in Y'$, then $y(j) \in \mathcal{Y}_j$.

Selecting such partitons for each j, $j = 1, 2, \ldots, r$ we have

$$|X'| \le \left(\prod_{1 \le j \le r} |\mathcal{X}_j| \right) \frac{n}{t^r},$$

and

$$|Y'| \le \left(\prod_{1\le j\le r} |\mathcal{Y}_j|\right)\frac{n}{t^r}.$$

Now since $|\mathcal{X}_j||\mathcal{Y}_j| \le (\frac{t}{2})^2$ for each j, $j = 1, 2, \ldots, r$, we obtain

$$|X'||Y'| \le \left(\frac{t^2}{4}\right)^r \left(\frac{n^2}{t^{2r}}\right) = \frac{n^2}{2^{1/\sqrt{\epsilon}}}. \tag{73}$$

Combining (71) and (73) yields

$$|X||Y| \le \frac{4n^2}{2^{1/\sqrt{\epsilon}}},$$

which establishes the theorem.

COROLLARY 5.2. *Given a positive constant ϵ and positive integer n there exists a bipartite graph $G = (U \cup V, E)$, $|U| = |V| = n$, such that if (X, Y) is any ϵ-regular pair, $X \subseteq U$, $Y \subseteq V$, with $d(X, Y) > \epsilon$ and $|X| = |Y| = m$, then*

$$m \le \frac{2n}{2^{1/2\sqrt{\epsilon}}}.$$

COROLLARY 5.3. *Given a positive constant ϵ and positive integer n, there exists a bipartite graph $G = (U \cup V, E)$, $|U| = |V| = n$, such that for each ϵ-regular partition $\mathcal{K}$ of $U \times V$ into cylinders we have*

$$|\mathcal{K}| > c2^{1/\sqrt{\epsilon}}, \quad \text{where } c \sim \frac{(1/\sqrt{e} - 2\epsilon)}{4}.$$

Proof. Let $G = (U \cup V, E)$, $|U| = |V| = n$ be the bipartite graph constructed in the proof of Theorem 5.1 and $\mathcal{K}$ an ϵ-regular partition of $U \times V$ into cylinders. A cylinder $X \times Y$, $X \subseteq U$, $Y \subseteq V$, is called "big" if

$$|X||Y| > 4n^2 2^{-1/\sqrt{\epsilon}},$$

and "small" otherwise.

Note that if $X \times Y$ is big, then by Theorem 5.1 the pair (X, Y) is either ϵ-irregular or has density at most ϵ. Consider the pairs (edges and nonedges) which are contained in big cylinders. These consist of at most ϵn^2 edges which are not in ϵ-regular cylinders, at most ϵn^2 edges which are in cylinders of density at most ϵ, and at most $(1 - d(U, V))n^2 \sim (1 - 1/\sqrt{e})n^2$ nonedges. Thus there are at least $\sim (1/\sqrt{e} - 2\epsilon)n^2$ pairs which are contained in small cylinders from which the corollary follows.

REFERENCES

[ADLRY] N. ALON, R. A. DUKE, H. LEFMANN, V. RÖDL, AND R. YUSTER, *The algorithmic aspects of the Regularity Lemma*, J. Algorithms, 16 (1994), pp. 80–109.

[CW] D. COPPERSMITH AND S. WINOGRAD, *Matrix multiplication via arithmetic progressions*, J. Symbolic Comput., 9 (1990), pp. 251–280.

[ER] N. EATON AND V. RÖDL, *Ramsey numbers for sparse graphs*, Emory University, Preprint.

[ELS] P. ERDŐS, L. LOVÁSZ, AND J. SPENCER, *Strong independence of graphcopy functions*, in L. Lovász, J. H. Spencer, eds., Graph Theory and Related Topics, Academic Press, New York, 1979, pp. 165–172 .

[FR] F. FRANEK AND V. RÖDL, *Ramsey problem on multiplicities of complete subgraphs in nearly quasirandom graphs*, Graphs Combin., 8 (1992), pp. 199–308.

[F] O. FRANK, *Estimating a graph from triad counts*, J. Statist. Comput. Simulation, 9 (1979), pp. 31–46.

[FH] O. FRANK AND F. HARARY, *Balance in stochastic signed graphs*, Social Networks, 2 (1979/80), pp. 155–163.

[NP] J. NEŠETŘIL AND S. POLJAK, *On the complexity of the subgraph problem*, Commentationes Mathematicae Universitatis Carolinae, 26 (1985), pp. 415–419.

[R] V. RÖDL, *On universality of graphs with uniformly distributed edges*, Discrete Math., 59 (1986), pp. 125–134.

[SS] M. SIMONOVITS AND V. T. SÓS, *Szemerédi's partition and quasirandomness*, Random Structures Algorithms, 2 (1991), pp. 1–10.

[Sz] E. SZEMERÉDI, *Regular partitions of graphs*, in *Proc. Colloque Inter. CNRS*, J.-C. Bermond, J.-C. Fournier, M. Las Vegas, and D. Sotteau, eds., 1978, pp. 399–401.

SIAM J. COMPUT.
Vol. 24, No. 3, pp. 621–649, June 1995

013

A VARIATIONAL METHOD FOR ANALYSING UNIT CLAUSE SEARCH*

HENRI-M. MÉJEAN†, HENRI MOREL†, AND GÉRARD REYNAUD†

Abstract. We expose a variational method for analysing algorithms, as applied to analyse the algorithm **UC**, which is the Davis–Putnam procedure for a set of clauses of three literals. The variable from a unit clause or, if there is none, the first remaining variable from a fixed list is chosen. The algorithm **UC** finds all the solutions as a set of cylinders. Following the nomenclature of Purdom [J. Inform. Process., 13 (1990), pp. 449–455], we call this algorithm "unit clause backtracking with cylinders of solutions."

First we give an expression for the number of nodes of the calculation trees of all the inputs. Then we use a variational method to calculate the base β of the principal exponential part of the average time of calculation T. This "exponential base" is the maximum of three elementary functions f_i of four real variables. These functions are defined on the product of the half positive real line by the 3-dimensional unit real cube. We finally obtain the following short statement. Let v be the number of the variables. Let c be the number of the clauses. Let $\gamma = \frac{c}{v} > 1$. Let γ be constant when v grows to infinity. The principal exponential part of the average time of **UC** is β^v where

$$\beta = \max_{0 \le \lambda \le 1} 2^{\lambda} \left(1 - \frac{3\lambda^2}{8} + \frac{\lambda^3}{4}\right)^{\gamma}.$$

We mean that $\lim_{v \to \infty} T^{1/v} = \beta$.

As a first consequence of our method we match **UC**, with the algorithm **B** without rearrangement (i.e., with a fixed order for introducing the variables). This gives a proof to a conjecture of P. W. Purdom [Artif. Intell., 21 (1983), pp. 117–133].

Key words. analysis of algorithms, computational complexity, satisfiability, variational methods

AMS subject classifications. 68, 49

Introduction. The problem of whether an "and of or's" F (conjunctive normal form) is satisfiable is known as the satisfiability problem. This problem is denoted by SAT, or 3SAT when each clause has 3 literals.

We consider an algorithm **UC** which solves 3SAT and finds all the solutions as a set of cylinders. We calculate the average number of nodes of a calculation tree for an input with a simple probability law [6].

The following works are close to ours: P. W. Purdom, Jr. [12], [13], [15] gives asymptotic results, a compact summation for the calculation time, and the conjecture that **UC** is in the difficult case exponentially better than **B**. The summation and conjecture were the main impulse for our work; moreover, the report that it was very difficult to give bounds to match **UC** and **B** gave us the idea of calculating exactly the base of the exponential part of the average time. See more in §1.1.

The works of J. Franco [3], Ming-Te Chao and J. Franco [10], A. Goldberg, P. W. Purdom, Jr., C. Brown, and E. Robertson [4], [14] give a probabilistic analysis of simplified Davis–Putnam procedures. B. Monien and E. Speckenmeyer [11] give an algorithm which solves SAT in less than 2^n steps. P. W. Purdom, Jr., and C. A. Brown [13] give upper and lower bounds for the average calculation time of **UC**. See more in a survey by P. W. Purdom [15]. V. Chvatal and E. Szemeredi [1] give an exponential lower bound for the size of the resolution proof of unsatisfiability for a fixed ratio of the number of clauses to the number of variables.

Our work gives a method for appreciating exactly the principal exponential part of the acceleration which is due to the choice of unit clauses.

*Received by the editors May 4, 1992; accepted for publication (in revised form) January 18, 1994.

†Laboratoire Analyse, Probabilités, Information, Faculté des Sciences, Luminy, case 901, 70 route Léon-Lachamp,13288 Marseille Cedex 9, France.

Let v be the number of the variables. Let c be the number of the clauses. Let $\gamma = \frac{c}{v} > 1$. Let γ be constant when v grows to infinity. Let

$$\beta_\gamma = \max_{0 \le \lambda \le 1} 2^\lambda \left(1 - \frac{3\lambda^2}{8} + \frac{\lambda^3}{4}\right)^\gamma. \tag{1}$$

The notation β_γ emphasizes that β_γ depends only on γ. We will write β for β_γ.

We prove that there exist an integer k_0 and two positive functions m_k and M_k of $k > k_0$ with limits zero when k grows to infinity, such that for every $k \ge k_0$, for every $\epsilon > 0$ we can find $v_{k,\epsilon}$ for which $v > v_{k,\epsilon}$ implies $\beta - m_k - \frac{\epsilon}{2} < T^{1/v} < \beta + M_k + \frac{\epsilon}{2}$.

Let k_1 be such that for $k \ge k_1$ we have $m_k < \frac{\epsilon}{2}$, $M_k < \frac{\epsilon}{2}$, then $v > v_{k_1,\epsilon}$ implies $\beta - \epsilon < T^{1/v} < \beta + \epsilon$; that is,

$$\lim_{v \to \infty} T^{\frac{1}{v}} = \beta.$$

(Thus, if we write $T = \beta^v h(v)$, $h(v)$ may be exponential but of less importance than β^v: for instance $h(v)$ could be $\beta^{\sqrt{v}}$.)

We will define a $\gamma_c = 4.5\ldots$ such that for $\gamma < \gamma_c$, β is $2(\frac{7}{8})^\gamma$; see §4.3, following "Remarks on the average number of solutions."

Our method uses variational arguments. For clarity, in this paper we give only asymptotic results and consider only the case $\gamma > 1$ (the case $\gamma \le 1$ needs slight modifications of our proofs).

1. Description of UC with input F.

1.1. Algorithm. At the end of the paper, there is an index of the first occurrence of each item.

Let v be integer ≥ 3.

The *literals* are the elements of a set L of $2v$ elements; on L we are given a partition P of v subsets of two elements. To each of these subsets $s = \{l, m\}$, $s \in P$, is associated a *variable* v_s which takes the value l or m. The number of the variables is v. If $s = \{l, m\}$ the two literals l and m are said to be in *opposition*, or *opposed*; we write $l = -m = -(-l)$.

An instance for 3SAT is a sequence $F = (c_i)_{1 \le i \le c}$, $c_i = \{x_i, y_i, z_i\}$, of *clauses* which are defined as sets of three (distinct) literals without opposition. (i.e., the literals of each clause have three different variables). Repeated clauses may be in F. Thus, c will be always the number of the clauses of the entry F we consider. The real number γ is the ratio $\frac{c}{v}$. An entry F is satisfied by $n = \{l_1, \ldots, l_p\}$, $p \le v$ iff for every $i = 1 \ldots c$, $c_i \cap n \ne \Phi$. If $i < i'$ we say that c_i is "before" $c_{i'}$.

Let $n = l_1 l_2 \ldots l_p$ be a list of p distinct literals without opposition. The set $\{l_1, l_2, \ldots, l_p\}$ is also denoted by n. The set $\{-l_1, \ldots, -l_p\}$ is denoted by $-n$. We refer to n as the "stack." The literals $l_1 \ldots l_p$ will be successively introduced by the algorithm.

For $i \le p$, the prefix $l_1 l_2 \ldots l_i$ of n is denoted by n_i. The set of clauses F_n is obtained from F by removing every clause satisfied by n and removing every literal of $-n$ in the remaining clauses. So, F is satisfied by n iff F_n is empty.

We define the following *lexicographic order* on the unit clauses δ of F_n. (The unique literal which is in the unit clause δ is also called δ.)

Let $r(\delta)$ be the smallest integer for which there is in F a clause u, $u = \{-l_j, -l_{r(\delta)}, \delta\}$, where $j < r(\delta)$, and let $r'(\delta)$ be the index of the first such u. Then $\delta < \delta'$ if either $r(\delta) < r(\delta')$ or $r(\delta) = r(\delta')$ and $r'(\delta) < r'(\delta')$. This is the condition for the value of δ being forced before δ' is assigned a value.

We are given a constant list $a_1 a_2 \ldots a_v$ of distinct literals without opposition. We say that this list is the *standard list*.

The algorithm **UC** is the following. We denote a state of the *stack* by n.

Begin. Let n be the empty sequence.
(A)
If there are two opposed unit clauses in F_n then
    this partial backtracking ends without solution;
else
    if there is a unit clause in F_n then
      choose as l_{p+1} the least one in the lexicographic order; go to (A);
    else if F_n is empty then
        this partial backtrack ends (n yields a cylinder of 2^{v-p} solutions);
      else let s be minimal such that a_s is neither in n nor in $-n$. Generate in parallel two
        calculations by taking a_s or $-a_s$ as l_{p+1}; let each calculation go to (A).

N.B. We choose this algorithm to analyse the importance of selecting forced literals. This is relevant for γ great enough, for instance, near the maximal entropy case (e.g., $4 < \gamma < 5$ for $v = 50$), i.e., when half the entries of c clauses with v variables are satisfied. So the algorithm **UC** is not interesting for low values of γ; for such a γ it would be useful to extend our analysis to an algorithm selecting literals in the shortest clauses, without using a standard list. See §5.

Let us call **U** the algorithm derived from **UC** by the following modification. The algorithm **U** does not stop on solution nodes n and lengthens such a node by a complete subtree.

We will use the fixed length clause distribution with equiprobability for each clause. This algorithm, and our probability model, are almost the same as they are in the analysis of [13]. If we replace in our version the test for F_n being empty by a test for all the variables being set we obtain the version of Purdom and Brown. Purdom analyses the same algorithm in [12]. All the techniques of algorithm **UC** are in the Davis–Putnam procedure [2], which also contains a number of additional techniques. So, for γ close to 1 where our algorithm is slow, do not conclude that the Davis–Putnam procedure is also slow.

The advantage of our method is particularly great when the number of clauses is proportional to the number of variables, a case of great importance for which the previous formulas did not give tight bounds. Our method gives more than tight bounds: it gives the exact principal exponential part.

1.2. The ordinary backtrack algorithm with a fixed order for the variables.

1.2.1. Description of B with input F. The algorithm **B** is the following. We denote a state of the stack by n.

Begin. Let n be the empty sequence.
(A)
If there is in F a clause $\{-l_i, -l_j, -l_p\}$, $i < j < p$ then
    this partial backtracking ends without solution;
else
    if F_n is empty then
        this partial backtrack ends (n yields a cylinder of 2^{v-p} solutions);
    else
        generate in parallel two calculations by taking a_{p+1} or $-a_{p+1}$ as l_{p+1};
        let each calculation go to (A).

1.3. An example of backtrack tree for UC and B. We give below an example of backtrack tree for **UC**. We also give for the same entry the backtrack tree for the algorithm **B**.

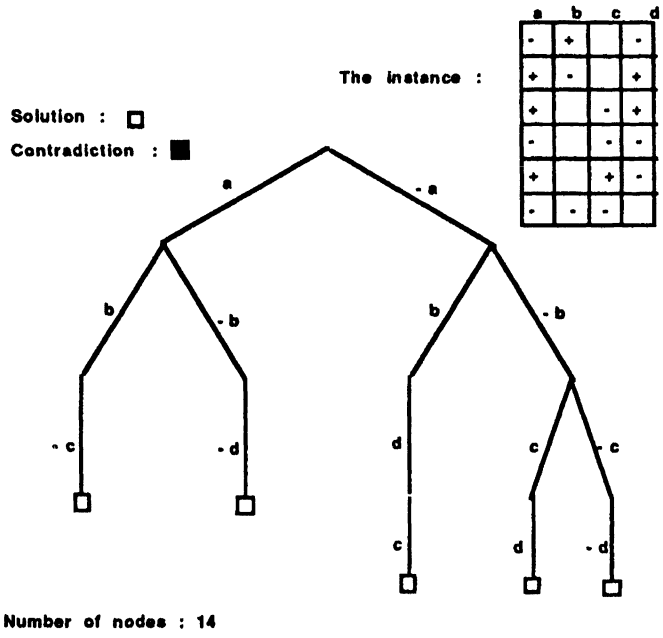


FIG. 1.

We see in Fig. 1 how **UC** works. On the second and third branch the forced literals $-d$ and d are introduced with rearrangement.

Figure 2 gives the calculation tree of **B** for the same instance F. These calculation trees are subtrees of the whole binary tree of depth v.

2. Notation for the analysis.

2.1. Nodes, status of the literals of the stack, selected clauses. Let $v \geq p \geq 3$. A state $n = l_1 l_2 \ldots l_p$ of the stack defines a node of the calculation tree that is built by **UC** from an F as a backtrack tree. If a node n is a prefix of a node n' we write $n \leq n'$.

We define two kinds of positions of literals in n: the *standard* positions and the *forced* literal positions. To each forced literal position i is associated a position $j < i$. The (position of the) literal l_j is called the *forcing* position for the (position of the) forced literal l_i. We have the following conditions for the positions i and j: the input F contains at least a clause $\{-l_{i'}, -l_j, l_i\}$, $i' < j < i$, which is selected using lexicographic order.

We define the *status* of a position i, $1 \leq i \leq p$ of a generated stack n. If l_i has been chosen as a forced literal of $F_{n_{i-1}}$, then the status of i is "forced" literal, or else it has been chosen as an a_s or $-a_s$ and the status of i is "standard."

We say that n is a binary node if in a next node nl_{p+1} of the backtrack tree l_{p+1} is standard, i.e., if F_n has no unit clause. Otherwise n is an unary node in the backtrack tree, or a terminal node.

Let M be the set of the positions in n with status "forced" and m be their number. Positions 1 and 2 are always standard. Thus $m \leq \min\{p-2, c\}$, where c is the number of the clauses.

Let $(j_s) = (j_s)_{1 \leq s \leq m}$ be a sequence of m distinct integers with $1 \leq j_s \leq c$. The clause c_{j_s} is selected by the algorithm to decide that an l_{b_s} is put into the stack as the sth forced literal. Thus c_{j_s} is the first clause in the corresponding lexicographic order which is not satisfied by n_{b_s-1} and contains the negations of two literals of n_{b_s-1}.

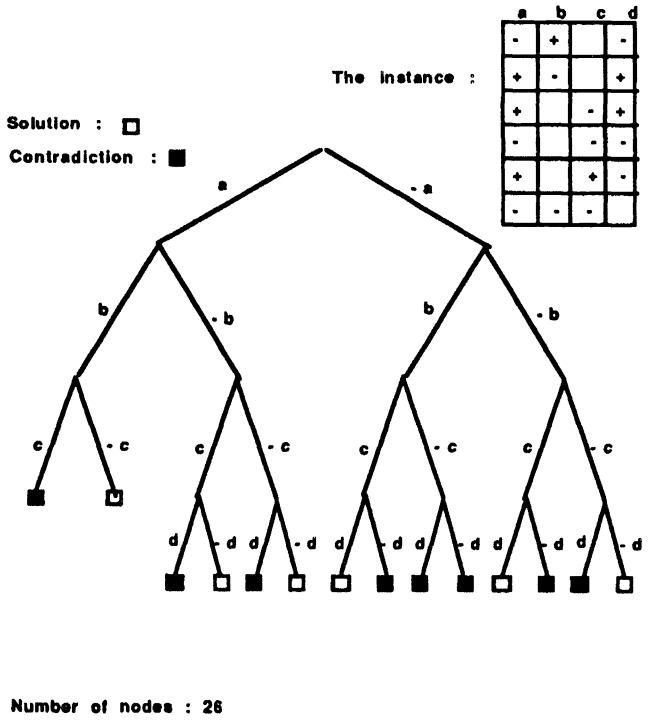


FIG. 2.

Let $(k_i) = k_1 k_2 \ldots k_m$ be the permutation of the integers $1, \ldots, m$ such that $j_{k_1} < j_{k_2} < \cdots < j_{k_m}$. Thus j_{k_1} is the index of the first selected clause $c_{j_{k_1}}$, and so on.

Consider again the definition of the lexicographic order in §1.1. If the unit clause literal $\delta = l_i$ has the position i in the stack n we also write r_i instead of $r(l_i)$ for the position of the forcing literal $l_{r(\delta)}$. For the clause $u = \{-l_j, -l_{r(\delta)}, \delta\}$ the two literals which are opposed to literals in the stack are said to be the *forcing* literals for the *forced* literal δ. By "forcing literal" we mean the second forcing literal unless stated otherwise.

In Fig. 1, the unit clause literal $-d$ is introduced by the selected clause $\{-a, b, -d\}$; we have now $n = (a, -b, -d) = (l_1, l_2, l_3)$; so $\delta = -d$, $r(\delta) = 2$, $j = 1$, $l_j = a$, $l_{r(d)} = -b$.

More generally, a literal $-l_{i'}$ is said to be the forcing literal of a clause $u = \{-l_i, -l_{i'}, x\}$ of an instance F iff $l_i, l_{i'}$ are two literals of n and $i < i'$. We will deal later with clauses in which the forcing literal of x was set before the position i'. If a selected clause is $c_{j_s} = \{-l_i, -l_{i'}, l_{b_s}\}$ where $i < i' < b_s$, we say that $-l_{i'}$ is the forcing literal of the sth forced literal l_{b_s} of n. (With the notation of §1.1 we have $i' = r(l_{b_s})$, $j_s = r'(l_{b_s})$.)

2.2. Moderated measures I, sets of nodes. The set of instances F is denoted by $\mathbf{F}$. There are $8\binom{v}{3}$ possible clauses, where $\binom{v}{3}$ is the usual binomial coefficient. Thus the cardinality of $\mathbf{F}$ is $|\mathbf{F}| = 8^c\binom{v}{3}^c$. We define subsets of $\mathbf{F}$ indexed in the following way. Let $\mathbf{F}_n$ be the subset of $\mathbf{F}$ from which **UC** generates the node n.

A literal l_i may be forcing for t_i forced literals, $t_i \geq 1$. The numbers t_i define a measure I with integral values on the interval of integers $[1, \ldots, p-1]$. The behaviour of these measures I is crucial in our method. We will detail the relation of I to the set of the forced positions as the "flattening" of I. The measures I must obey the so-called constraints of "moderation." These measures I will be used to index the sets $\mathbf{F}_{n,I}$ of a partition of the set $\mathbf{F}_n$ of the instances.

So, let $I = (t_i)_i = (t_i)_{i=2,\ldots,p-1}$ be a sequence of integers with $t_i \geq 0$; t_i will be the number of variables which are forced when variable i is set. We set $m = m_I = \sum_{2\leq i\leq p-1} t_i$.

Let $\mathbf{F}_{n,I}$ be the subset of $\mathbf{F}_{n,I}$ from which **UC** generates n in such a way that for $2 \leq i \leq p-1$, t_i is the number of forced literals of n which have $-l_i$ as forcing.

The sets $\mathbf{F}_{n,I}$ are empty if the indices are not compatible in the sense studied in the next section. The fact that $n = l_1 \ldots l_{p-1}$ is a binary node implies that l_p is not forced. (Otherwise, the set $\mathbf{F}_{n,I}$ would be empty). So I must satisfy the following necessary conditions:

(2) for every $r \in [2, \ldots, p-1]$, $I([r, p-1]) = t_r + t_{r+1} + \cdots + t_{p-1} < p - r$.

If I satisfies (2) we say that I is *moderated* on $[2, p-1]$; it implies that $t_{p-1} = 0$.

We will prove in the next section that condition (2) is *sufficient* in the following sense. If (2) is true, the status that **UC** calculates from $F \in \mathbf{F}_{n,I}$ for each l_j of n may be calculated only from I (in such a way that every t_i is the number of the forced literals l_j whose forcing literal is $-l_i$).

2.3. Calculation of the status from any moderated measure I on $[2, p-1]$. We will also define and calculate from I an integer $q = q(I)$.

In the case that $t_i = 0$ for $2 \leq i \leq p-1$, the l_j are standard for $j = 1, \ldots, p$. So we let $q = 0$ in this case. Otherwise let $m_1 - 1$ be the least i such that $t_i \neq 0$. Then $l_1, \ldots, l_{m_1-1}$ are standard.

If $t_{m_1-1} + \cdots + t_{p-1} = p - m_1 + 1$ the l_i are forced for $m_1 \leq i \leq p$. We let $q = 1$ in this case. Else let m_2 be the least integer such that $t_{m_1-1} + \cdots + t_{m_2-1} < m_2 - m_1 + 1$.

So m_2 is the integer such that $m_2 - m_1$ is the length of the first round of forced literals. Then l_{m_2} is the first literal of a second segment of standard literals. If there is no $i > m_2$ with $t_i > 0$ let $q = 2$. Otherwise let m_3 be the least such i. We may iterate and calculate q and the sequence $2 < m_1 < \cdots < m_q < p$ such that the status is as follows:

(3) $$\overbrace{l_1 l_2 \ldots l_{m_1-1}}^{\text{st}}\ \overbrace{l_{m_1}\, l_{m_1+1} \ldots l_{m_2-1}}^{\text{forced}}\overbrace{l_{m_2}\, l_{m_2+1} \ldots l_{m_3-1}}^{\text{st}} \ldots\ldots \overbrace{l_{m_q}\, l_{m_q+1} \ldots l_p}^{\text{st}^{q+1}},$$

where *st* means "standard." If q is odd *st*^{q} means "standard" else it means "forced."

Let $M = M_I$ denote the union of all the intervals $[m_1, m_2[$, $[m_3, m_4[$, ... which are so calculated and marked as forced. Let q' denote the number of these intervals which have M_I as union. We call M_I the *flattening* of I. We also say that I flattens on M_I. If $t_i \leq 1$ for every i, we say that I is flat. Indeed, the flattening M_I is the same as $\underline{I}$ (the support of I) shifted right by 1.

So the sequence of intervals above appears when flattening I, and it is determined if only I is known, independently of the literals l_i of the stack n. In the intervals labeled as standard, the literals l_i may be any literals which are not in n_i or $-n_i$, i.e., any literals l_i which for every $i' < i$ are different from both l_i and $-l_{i'}$. In the intervals labeled as standard the literal l_i must be the first in the standard list which is not in n_i or $-n_i$.

So we say that a node n is *compatible* with I if every literal l_i of $n - M_I$ is a_s or $-a_s$, where s is the least integer such that a_s and $-a_s$ are not in n_{i-1}. There are $2(v - i + 1)$ possibilities for the forced literal l_i at position i in n. Let $m = \mid M \mid$.

The number of nodes which are compatible with I and have length p is

(4) $$c_I = 2^p \prod_{i\in M} (v - i + 1).$$

If M is empty $c_I = 2^p$.

If q is even the literals $l_{m_q}, \ldots, l_p$ become standard by the preceding calculation. Then we have $t_{m_q-1} = t_{m_q} = \cdots = t_{p-1} = 0$ and condition (2) is sufficient.

$$(5) \qquad \overbrace{l_1 l_2 \ldots l_{m_1-1}}^{\text{st}} \overbrace{l_{m_1} l_{m_1+1} \ldots l_{m_2-1}}^{\text{forced}} \overbrace{l_{m_2} l_{m_2+1} \ldots l_{m_3-1}}^{\text{st}} \ldots\ldots \overbrace{l_{m_q} l_{m_q+1} \ldots l_p}^{\text{st}}.$$

We see that q is even iff the measure $I = (t_i)_i$ is moderated on $[2, p-1]$. We get $q' = \frac{q}{2}$.

3. Calculation of the number of binary nodes of length $p-1$ which are generated by UC. We will not examine the case of q odd here. We suppose now that $q(I)$ is even and so we calculate the total number of length p nodes which are generated from $\mathbf{F}_{n,I}$ by **UC**. The integer q is even iff the last literal l_p of n is standard, i.e., iff the node n_{p-1} is binary. Thus we will calculate the total number of length p nodes which are generated from $\mathbf{F}$ by **UC** and such that their preceding node is binary. Thus, our final results are affected by a polynomial factor, i.e., are not modified.

We first calculate a summation for the exact number of binary nodes of all the backtrack trees. Then we divide the summation into a polynomial number of portions and use a variational argument to obtain the result.

3.1. Sets of clauses which are defined with the stack n. In the following text until (8) we let I represent the essential information about the nature of the stack. As a function of I, we calculate how many predicates would result in the program producing a stack of the kind specified by I. To do the calculation we need to compute how many clauses exist of various types.

We are given $n = \{l_1, l_2, \ldots, l_{p-1}\}$ and v. We will consider that the possible clauses that occur in an entry are of the following four different types:

1) the clauses u which contain at most one literal of $-n = \{-l_1, -l_2, \ldots, -l_{p-1}\}$,

2) the clauses which contain $-l_k$ as forcing literal, $k = 3, \ldots, p-1$, and contain an l_j with $j < k$.

3) the clauses which are in the sets $S_{j,k}$ defined below which contain a forced literal l of the stack, the forcing literal of l, and another literal of $-n$.

4) the clauses of the set $\mathbf{R}_I$ which are defined below.

We call G_p the union of the clauses of type 1 and 2.

The clauses of type 1, 2, and 4 can be inserted everywhere when building an entry for a given n, without changing n. We count separately the clauses of type 4 because we use for that the appropriate ρ_i which will be further defined. The clauses of type 3 cannot be inserted anywhere.

After the necessary definitions, we will check in §3.2 that no type of clause has been omitted.

We set $g_p = \mid G_p \mid$. We get, after classic calculation,

$$(6) \qquad g_p = \frac{4}{3}v(v-1)(v-2) - v(p-1)(p-2) + \frac{2}{3}p(p-1)(p-2).$$

Let G_p^* be the subset of the clauses of G_p satisfied by $n = \{l_1, l_2, \ldots, l_{p-1}\}$. We set $g_p^* = \mid G_p^* \mid$. We get

$$(7) \qquad g_p^* = 2(p-1)v^2 - (p-1)v(p+4) + 2p(p-1).$$

g_p is calculated as follows. The total number of possible clauses is $2^3 v(v-1)(v-2)/3!$. This is the first term in the formula for g_p. The number g_p = number of all clauses -

number of clauses $\{-l_i, -l_j, x\}$, $x \notin n$, $x \notin -n$, - number of clauses $\{-l_i, -l_j, l_k \text{ or } -l_k\}$, $i < j < k \leq p-1$. So,

$$g_p = \frac{4}{3}v(v-1)(v-2) - (p-1)(p-2)(v-p+1) - \frac{1}{3}(p-1)(p-2)(p-3).$$

Let I be a moderated measure on $[2, p-1]$. Let $\underline{I} = \{i_1, \ldots, i_\sigma\}$ be the support of I, $i_1 < \cdots < i_\sigma$, (i.e., for $j = 1, \ldots, \sigma$, $t_{i_j} > 0$ and $t_i = 0$ if $i \notin \underline{I}$). Every F in $\mathbf{F}_{n,I}$ is a *shuffle* of sequences of the following type.

For $j = 1, \ldots, \sigma$, $k = 1, \ldots, t_{i_j}$, let $S_{j,k}$ be the nonempty subsequence of clauses of F which contains the negations $-x$,$-l_{i_j}$ of two literals of n and the kth forced literal in n for which $-l_{i_j}$ is the forcing literal. (The algorithm **UC** selects the first clause of $S_{j,k}$ to generate this forced literal and later finds that the following clauses of $S_{j,k}$ are thus satisfied.) Let $\mu_{j,k}$ be the length of $S_{j,k}$ and $\mu_j = \sum_{1 \leq k \leq t_j} \mu_{j,k}$. Let $\mu = \sum_{1 \leq j \leq \sigma} \mu_j$.

Definition of ρ_i. Let p and any moderated measure I on $[2, p-1]$ be given. For any position i in the stack let ρ_i be the number of the forced literals l_k, $i < k$, the forcing literal of which has a position $i' < i$.

We see that $\rho_i \geq t_{i-1} - 1$; $\rho_i = \sum_{j_0 \leq j < i} t_j - \theta$ where j_0 is the position of the forcing literal for the forced literal l_{i+1}, and θ is the number of the literals l_s, $s \leq i$, which are forced by l_{j_0}.

Definition of r_I. Let $\mathbf{R}_I$ be the set of the clauses $\{-l_j, -l_i, l_k\}$, $j < i$, which are satisfied by a forced literal l_k which is generated with a forcing literal $l_{i'}$, $i' < i$ and has its index k in $]i, i + \rho_i]$.

Let $r_I = |\ \mathbf{R}_I\ | = \sum_{i \in [2, p-1]} (i-1)\rho_i$. Let W be the subsequence of F of the clauses which are in R_I.

3.2. Entries as shuffles. Let $w = |\ W\ |$. Let $s = w + \mu$. Thus, F is a shuffle of these $S_{j,k}$, of W and of the subsequence S of the $c - s$ remaining clauses which are in G_p. These integers depend on I, which is taken to be an index for r_i and ρ_i.

Now we check that there is no other type of clause to consider to build an entry which admits n as a node of its backtrack tree and I as the distribution of the forcing literals.

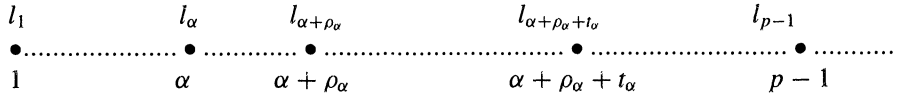
Let $u = \{x, y, z\}$ be a clause in such an entry. If no more than one of the literals x, y, z is in $-n$, this clause is considered in G_p. Moreover x, y, z cannot be all in $-n$ *because n is a binary node.*

Now, we have only to consider the clauses u which have two literals in $-n$; we can write $u = \{-l_\beta, -l_\alpha, z\}$, $\beta < \alpha$. The literal z must be in n because, if z had not been forced previously by another suitable clause, z was forced by this clause u. Thus, $u = \{-l_\beta, -l_\alpha, l_\gamma\}$, and we suppose $\gamma > \alpha$ otherwise u would be in G_p.

If u is the selected clause which introduces l_γ in n as forced, u is in a $S_{j,k}$. Therefore it has been considered.

Consequently we suppose that l_γ is introduced by a clause other than u. (Thus the forcing for l_γ is l_α or on the left of l_α.)

Consider the following figure for $n = l_1 \ldots l_{p-1}$:



We recall that ρ_α is the number of all the forced literals on the right of α which are forced by literals on the left of α; these forced literals range from $\alpha + 1$ to $\alpha + \rho_\alpha$. On the right of $\alpha + \rho_\alpha$ there are t_α forced literals with l_α as forcing; on the right of $\alpha + \rho_\alpha + t_\alpha$, further forced

literals are possible, but their forcing literals must be on the right of α. This implies that the literal l_γ of u is on the left of $\alpha + \rho_\alpha + t_\alpha$: $\gamma \leq \alpha + \rho_\alpha + t_\alpha$.

We have two cases:

1) if $\alpha + \rho_\alpha < \gamma$, l_γ is forced by l_α, so $u = \{-l_\beta, -l_\alpha, l_\gamma\}$ is in an $S_{j,k}$; otherwise

2) $\alpha < \gamma \leq \alpha + \rho_\alpha$, which implies $\rho_\alpha > 0$; we see that u is in $\mathbf{R}_I$.

This checking emphasizes the different parts played by the sets $S_{j,k}$ and $\mathbf{R}_I$; these two types of sets are relative to the structure of the lexicographic order.

3.3. Total number $D_{I,p-1}$ of the binary nodes of length $p-1$ which the algorithm UC generates according to a measure I. A node is counted as many times it occurs in the different backtrack trees. Given I, every clause of the sequence $S_{j,k}$ may be chosen in $u_j = i_j - 1$ different ways. Every clause of W may be chosen in r_I different ways. Then the total number of the shuffles of the sequences above is for given s, w, I,

(8)

$$\underline{X}(s) = \underline{X}(s, w, I)$$

$$= g_p^{c-s}(r_I)^w \sum_{\mu=\mu_{1,1}+\ldots+\mu_{1,t_{i_1}}+\ldots+\mu_{\sigma,1}+\ldots+\mu_{\sigma,t_{i_\sigma}}} \binom{c}{c-s, \mu_{1,1}, \ldots, \mu_{1,t_{i_1}}, \ldots\ldots, \mu_{\sigma,1}, \ldots, \mu_{\sigma,t_{i_\sigma}}, w} \cdot \prod_{1\leq j\leq\sigma} u_j^{\mu_j},$$

where $\mu_{j,k} \geq 1$, $1 \leq j \leq \sigma$, $1 \leq k \leq t_{i_j}$.

We first analyse **U**. (See N.B. in §1.1. **U** is the algorithm derived from **UC** by the following modification: **U** does not stop on solution nodes and lengthens such a node by a complete subtree. We may either suppose $p \leq v - 1$ or consider for a while that we also count length v solution nodes which are not binary.)

Let n be given and take any shuffle F above of W and the $S_{j,k}$; then **U** calculates the right forced literals, except for a permutation in each subset of the t_j forced literals which have the same forcing literal l_{i_j}. The number of shuffles from which **U** generates such a permutation of n does not depend on this permutation. We will do a division by $\prod_i t_i!$ in the following expression.

We recall that this summation is a variant of Purdom's summation in [13, p. 721]. Here we consider forced literals as introduced one by one, not by batches. We emphasize the particular set W of clauses.

In the preceding formula, informally speaking, the u's and μ's relate to those clauses that force variables; the r_I and w relate to clauses that would have forced variables except that the value of the variable was already known by the time the forcing would have occured. The g_p and $c - s$ relate to the clauses that don't do any forcing.

The integer t_j is the number of variables that are forced when j is set; $\mu_{j,k}$ is the number of clauses that force variable k when the forcing literal j is set.

Let I be moderated on $[2, p-1]$ with $t_{p-1} = 0$. We denote by $D_{I,p-1}$ (resp., $\underline{D}_{I,p-1}$) the total number of binary nodes n of length $p-1$ which **UC** (resp., **U**) generates according to I, each node being counted as many times it occurs in a backtrack tree. So,

$$2\underline{D}_{I,p-1} = \sum_{0\leq s\leq c} \frac{c_I \underline{X}(s)}{\prod_{1\leq j\leq\sigma} t_{i_j}!} = \sum_{0\leq s\leq c} \frac{c_I \underline{X}(s)}{\prod_{2\leq i\leq p-1} t_i!}. \tag{9}$$

We now return to the analysis of **UC**.

We will first study $D_{I,p-1}$ to obtain the average number T of binary nodes of length p, which will be

$$T = \frac{D_p}{8^c \binom{v}{3}^c},$$

with $D_p = \sum_I D_{I,p}$, where the summation runs over all the I which are moderated on $[2, p-1]$.

We take off the nodes which **U** generates although they are satisfied by $l_1, l_2, \ldots, l_{p-1}$.

Thus, we get $D_{I,p-1}$:

$$2D_{I,p-1} = c_I \left(\frac{1}{\prod_{1 \le j \le \sigma} t_{i_j}!} \right) \sum_{s,w} P(I,\mu) \binom{c}{s} \binom{s}{w} (r_I)^w \left(g_p^{c-s} - g_p^{*c-s} \right), \tag{10}$$

where $\mu = s - w$, where s runs over $[m, c[$, w runs over $[0, s-m]$, and where $P(I,\mu)$ is

$$P(I,\mu) = \sum_{\mu = \mu_{1,1} + \ldots + \mu_{1,t_{i_1}} + \ldots + \mu_{\sigma,1} + \ldots + \mu_{\sigma,t_{i_\sigma}}} \binom{\mu}{\mu_{1,1}, \ldots, \mu_{1,t_{i_1}}, \ldots\ldots, \mu_{\sigma,1}, \ldots, \mu_{\sigma,t_{i_\sigma}}} \prod_{1 \le j \le \sigma} u_j^{\mu_j}, \tag{11}$$

where the summation runs over every decomposition of μ as $\mu = \sum_{1 \le j \le \sigma} \mu_j$ and the decomposition of each μ_j as $\mu_j = \sum_{1 \le k \le t_{i_j}} \mu_{j,k}$, with all the μ_j, $\mu_{j,k} \ge 1$. We emphasize that $m \le s < c$; $s = c$ is impossible since n is binary. On the formula (10) we see that $D_{I,p-1} = 0$ if $c = s$.

We will now study $P(I,\mu)$.

Let $x_1, \ldots, x_m$ be real positive numbers (it is not useful here to consider a more general case).

Let

$$(x_1 + \cdots + x_m)^{\backslash \mu} = \sum_{\mu = \mu_1 + \ldots + \mu_m, \mu_i \ge 1 \text{for } 1 \le i \le m} \binom{\mu}{\mu_1, \ldots, \mu_m} \prod_i x_i^{\mu_i}. \tag{12}$$

(We use the notation $\backslash \mu$ because we think of the expression above as a "diagonal power," because every $\prod_i x_i^{\mu_i}$ contains the diagonal product $\prod_i x_i$ as a factor.)

So we consider that $P(I,\mu) = (x_1 + \cdots + x_m)^{\backslash \mu}$, where the list $x_1, \ldots, x_m$ is the list

$$\overbrace{u_1, \ldots, u_1}^{t_{i_1} \text{ terms}} \ \overbrace{u_2, \ldots, u_2}^{t_{i_2} \text{ terms}} \ \ldots, \overbrace{u_\sigma, \ldots, u_\sigma}^{t_{i_\sigma} \text{ terms}};$$

thus,

$$x_1 + \cdots + x_m = t_{i_1} u_1 + \cdots + t_{i_\sigma} u_\sigma = \sum_{2 \le i \le p-1} t_i (i-1).$$

(The notation $(\sum_{2 \le i \le p-1} t_i (i-1))^{\backslash \mu}$ supposes that the sum inside has formally m terms.)

We see that

$$P(I,\mu) = 0 \quad \text{if } \mu < \sum_{2 \le i \le p-1} t_i = m \qquad \text{and} \qquad P(I,m) = m! \prod_{2 \le i \le p-1} {u_i}^{t_i}.$$

As a consequence of the definition of $(x_1 + \cdots + x_m)^{\backslash \mu}$ we have $(x_1 + \cdots + x_m)^{\backslash \mu} \le (x_1 + \cdots + x_m)^{\mu}$.

Thus,

$$P(I,\mu) \leq \left(\sum_{2 \leq i \leq p-1} t_i (i-1) \right)^{\mu}.$$

We will use the following bound:

$$P(I,\mu) \leq \frac{\mu!}{(\mu - m + 1)! m!} P(I,m) H_I^{\mu - m}, \tag{13}$$

where

$$H_I = \sum_{2 \leq i \leq p-1} t_i (i-1). \tag{14}$$

Proof of the preceding inequality. Let m be an integer ≥ 1. Let $\mu, u_1, \ldots, u_m$ be real numbers with $\mu \geq m$, $u_i \geq 0$ for $i = 1, \ldots, m$.

Let

$$M_{m,\mu} = \left\{ (\mu_1, \ldots, \mu_m),\ \mu_i \text{ real numbers } \geq 1 \text{ with } \sum_{1 \leq i \leq m} \mu_i = \mu \right\}.$$

It is easy to prove first that for $(\mu_1, \ldots, \mu_m) \in M_{m,\mu}$

$$\prod_{1 \leq i \leq m} \mu_i \geq \mu - m + 1. \tag{15}$$

(Roughly speaking, we will get closer to the result by replacing inside the product $\prod_i \mu_i$ two factors x and y, $x > 1$, $y > 1$, by 1 and $y + x - 1$. Thus the sum of these factors has not changed; their product has decreased.) □

Then we prove that

$$(u_1 + \cdots + u_m)^{\backslash \mu} \leq \frac{\mu!}{m!(\mu - m + 1)!} (u_1 + \cdots + u_m)^{\backslash m} (u_1 + \cdots + u_m)^{\mu - m}.$$

This is equivalent to

$$\sum_{M_{m,\mu}} \frac{\mu!}{\mu_1! \ldots \mu_m!} \prod_{1 \leq i \leq m} {u_i}^{\mu_i}$$

$$\leq \frac{\mu!}{m!(\mu - m + 1)!} m! \prod_{1 \leq i \leq m} u_i \sum_{M_{m,\mu}} \frac{(\mu - m)!}{(\mu_1 - 1)! \ldots (\mu_m - 1)!} \prod_{1 \leq i \leq m} {u_i}^{\mu_i - 1},$$

which is equivalent to

$$\sum_{M_{m,\mu}} \frac{1}{\prod_{1 \leq i \leq m} \mu_i} \frac{\prod_{1 \leq i \leq m} {u_i}^{\mu_i}}{(\mu_1 - 1)! \ldots (\mu_m - 1)!} \leq \sum_{M_{m,\mu}} \frac{1}{\mu - m + 1} \frac{\prod_{1 \leq i \leq m} {u_i}^{\mu_i}}{(\mu_1 - 1)! \ldots (\mu_m - 1)!}.$$

The last inequality is a consequence of (15).

Remark. if we do not use (15) we get immediately from the last two lines that

$$P(I,\mu) \leq \frac{\mu!}{(\mu - m)! m!} P(I,m) H_I^{\mu - m}.$$

This result will satisfy the requirements of §4.3, unless we wanted to evaluate exactly the degree of the polynomial factor, which we will ignore.

4. Asymptotic behaviour of the average number of nodes. We will proceed in nine successive steps.

4.1. Notation. We let $\gamma = \frac{c}{v}$, $\sigma = \frac{s}{v}$, $\lambda = \frac{p}{v}$ and, as in §4.3, μ will change sense with $\mu = \frac{m}{v}$. We will shorten by supposing $\gamma > 1$.

If the two quantities x and y that we consider below grow infinitely when p and v do so and x/y has the limit 1, we say that $x \sim y$.

We have the following behaviour of g_p. Let $p = v(1 - \eta(v)) = v(1 - \eta)$.

We also have $8\binom{v}{3} \sim 4v^3/3$. Thus, $T^{1/v} \sim (D_p/(4v^3/3)^c)^{1/v}$.

We will prove that g_p and g_p^* can be replaced by the following values:

$$g_p \sim \frac{4}{3}v^3 D(\lambda), \qquad \text{where } D = D(\lambda) = 1 - \frac{3\lambda^2}{4} + \frac{\lambda^3}{2}$$

and

$$g_p^* \sim \frac{4}{3}v^3 N(\lambda), \qquad \text{where } N = N(\lambda) = \frac{3}{2}\left(\lambda - \frac{\lambda^2}{2}\right).$$

We will write now g for g_p and g^* for g_p^*.

Suppose $\lim_{v\to\infty} \eta = 0$. Therefore $g^*/g - 1 = -2\eta/v - 2\eta^2 + 4\eta^2/v + 8\eta/3v^2 + 2\eta^3/3 +$ higher-order terms.

Note that, for instance, η/v^2 may be of lower order than η^2/v. We are allowed to set these definitions, although p and v are two different variables, because $0 < p < v$ and because of the special form of the functions we consider, such as g_p.

4.2. Asymptotic behaviour for UC. We set

$$c_I = 2^p c_I'.$$

We recall that m is an abbreviation of $m_I = |\ I\ | = \sum_{2\le i\le p-1} t_i$. The average time for **UC** is polynomially (in fact, within a factor of v) equivalent to

$$T = \max_p \left(2^p \sum_{I,\mu,w} Q_{s,w}\left(1 - \left(\frac{g^*}{g}\right)^{c-s}\right)\right), \tag{16}$$

where $s = \mu + w$, μ runs over $[m, c[$, w runs over $[0, c - \mu[$, and

$$Q_{s,w} = c_I' \frac{1}{\prod_{2\le i\le p-1} t_i!} \frac{P(I,\mu)}{(4v^3/3)^c} \binom{c}{s}\binom{s}{w} (r_I)^w g^{c-s}.$$

4.3. Elimination of the parameters μ and w by a polynomial bound and summation, dropping the factor $(1-(g^*/g)^{c-s})$. We will match $Q_{s,w}$ with $Q_{s',w}$, where $s' = m + w$, i.e., in the limit case $\mu = m$.

Consider $0 \le m \le v < c$ and $0 \le w < c - m$ as fixed; then the possible s verify $c > s \ge s'$ with $s' = m + w$.

We have

$$\frac{Q_{s,w}}{Q_{s',w}} = \frac{1}{g^{\mu-m}} \frac{P(I,\mu)}{P(I,m)} \frac{(c-w-m)!}{(c-w-\mu)!} \frac{m!}{\mu!}.$$

We may replace above $g^{\mu-m}$ by $(4Dv^3/3)^{\mu-m}$ within a constant factor when $v \to \infty$.

Indeed, from the expression of g_p we obtain $g/(4v^3/3) = D - \frac{3}{v} + (1/2v^2) + \frac{9}{4}\frac{\lambda}{v} + (\lambda/v^2) - \frac{3}{2}\frac{\lambda}{v}$, where $\lambda = p/v$. Thus, $g/(4v^3/3) = D - \frac{3}{v} + \frac{3}{4}\frac{\lambda}{v} + \frac{\epsilon(v)}{v}$; as $\frac{3}{4} \le D \le 1$,

$0 \le \mu - m \le c$, $E = (g/(4Dv^3/3))^{\mu-m} = \left(1 - \frac{3}{D}\frac{1}{v} + \frac{\epsilon(v)}{v}\right)^{\mu-m}$. So, for v large enough, $\left(1 - \frac{4}{v} + \frac{\epsilon(v)}{v}\right)^{\mu-m} \le E \le 1$ as $\lim_{v\to+\infty}\left(1 - \frac{4}{v} + \frac{\epsilon(v)}{v}\right)^{c} = e^{-4\gamma}$. Our assertion is proved.

Now we apply (13):

$$\frac{Q_{s,w}}{Q_{s',w}} \le \frac{1}{(4D\,v^3/3)^{\mu-m}} H_I^{\mu-m} \frac{(c-w-m)(c-w-m-1)\dots(c-w-\mu-1)}{(\mu-m+1)!}.$$

We will use the following facts:

$$H_I = \sum_{2\le i\le p-1} t_i(i-1) \le \frac{p^2}{2} \le \frac{v^2}{2},$$

$$D \ge \frac{3}{4},$$

$$\frac{(c-w-m)(c-w-m-1)\dots(c-w-\mu-1)}{v^{\mu-m}} \le \gamma^{\mu-m},$$

where $\gamma = c/v$. From the term v^3 inside $(4D\,v^3/3)^{\mu-m}$, we associate $v^{2(\mu-m)}$ to simplify with the bound $v^2/2$ of H_I and $v^{\mu-m}$ to $(c-w-m)(c-w-m-1)\dots(c-w-\mu-1)$ and we obtain

$$\frac{Q_{s,w}}{Q_{s',w}} \le \frac{(\gamma/2)^{\mu-m}}{(\mu-m+1)!}.$$

Let $d(\gamma)$ be the integral value of x such that $(\gamma/2)^x/(x+1)!$ is maximal and let $(\gamma/2)^{d(\gamma)}/\big(d(\gamma)+1\big)! = a(\gamma)$.

(Thus, $d(\gamma)+1$ is the integral part $[\frac{\gamma}{2}]$ of $\frac{\gamma}{2}$). If γ is bounded, $Q_{s,w} \le a(\gamma)Q_{s',w}$ uniformly when p, m, and w vary. Thus, if γ is bound, $Q_{s,w} \le a(\gamma)Q_{s',w}$ uniformly when p, m, and w vary. Therefore we are allowed to replace $Q_{s,w}$ by $Q_{s',w}$, $(s' = m + w)$.

Now we sum over w; the variable w will no longer appear in our expression for T.

We will also use the fact that $s \ge s'$ implies that

$$1 - \left(\frac{g^*}{g}\right)^{c-s} \le 1 - \left(\frac{g^*}{g}\right)^{c-s'};$$

this implies that for all the considered values of s,

$$Q_{s,w}\left(1 - \left(\frac{g^*}{g}\right)^{c-s}\right) \le a(\gamma)Q_{s',w}\left(1 - \left(\frac{g^*}{g}\right)^{c-s'}\right)$$

and, since $s \le c$,

$$Q_{s',w}\left(1 - \left(\frac{g^*}{g}\right)^{c-s'}\right) \le \sum_s Q_{s,w}\left(1 - \left(\frac{g^*}{g}\right)^{c-s}\right) \le c\,a(\gamma)Q_{s',w}\left(1 - \left(\frac{g^*}{g}\right)^{c-s'}\right).$$

Thus, we can replace within a factor of c in the expression of T,

$$\sum_{s,w} Q_{s,w}\left(1 - \left(\frac{g^*}{g}\right)^{c-s}\right)$$

by

$$\sum_w Q_{m+w,w}\left(1 - \left(\frac{g^*}{g}\right)^{c-m-w}\right),$$

where w ranges in $[0, c-m]$.

We may write the following expressions

$$T =_1 \max_p 2^p \sum_{I,m,w} Q_{m+w,w} \left(1 - \left(\frac{g^*}{g}\right)^{c-m-w}\right),$$

$$T =_3 \max_p 2^p \max_{m,\mu} \sum_{|I|=m,w} Q_{m+w,w} \left(1 - \left(\frac{g^*}{g}\right)^{c-m-w}\right).$$

(By $=_k$ we mean equality within one polynomial factor of degree k in c and v.)
We rewrite $Q_{m+w,w}$ by replacing $P(I, m)$ by $m! \prod_i u_i^{t_i}$ and simplifying by $m!$:

$$Q_{m+w,w} = c'_I \frac{1}{(4v^3/3)^m} \left(\prod_i \frac{u_i^{t_i}}{t_i!}\right) \frac{c!}{(c-m-w)!w!} \frac{(r_I)^w}{(4v^3/3)^{c-m}} g^{c-m-w}.$$

Therefore

$$\sum_w Q_{m+w,w} \left(1 - \left(\frac{g^*}{g}\right)^{c-m-w}\right)$$

$$= c'_I \frac{1}{(4v^3/3)^m} \left(\prod_i \frac{u_i^{t_i}}{t_i!}\right) \frac{c!}{(c-m)!} \sum_w \binom{c-m}{w} \frac{(r_I)^w}{(4v^3/3)^{c-m}} (g^{c-m-w} - g^{*c-m-w}).$$

Now,

$$\sum_w Q_{m+w,w} \left(1 - \left(\frac{g^*}{g}\right)^{c-m-w}\right)$$

$$= c'_I \frac{1}{(4v^3/3)^m} \left(\prod_i \frac{u_i^{t_i}}{t_i!}\right) \frac{c!}{(c-m)!} \left(\left(\frac{g}{(4v^3/3)}\right)^{c-m} \left(1 + \frac{r_I}{g}\right)^{c-m} - \left(\frac{g^*}{(4v^3/3)}\right)^{c-m} \left(1 + \frac{r_I}{g^*}\right)^{c-m}\right).$$

So we may write for the average time T

$$T = \max_p \max_{m \le p} \sum_{|I|=m} (Q_I - Q'_I), \tag{17}$$

where

$$Q_I = 2^p \left(\frac{3}{4}\right)^m \frac{c!}{(c-m)!} \left(\frac{g}{4v^3/3}\right)^{c-m} \mathbf{M}_I \frac{1}{v^{3m}} c'_I \prod_i \frac{u_i^{t_i}}{t_i!}, \tag{18}$$

$$Q'_I = 2^p \left(\frac{3}{4}\right)^m \frac{c!}{(c-m)!} \left(\frac{g^*}{4v^3/3}\right)^{c-m} \mathbf{M}'_I \frac{1}{v^{3m}} c'_I \prod_i \frac{u_i^{t_i}}{t_i!}$$

with

$$\mathbf{M}_I = \left(1 + \frac{r_I}{g}\right)^{c-m}, \qquad \mathbf{M}'_I = \left(1 + \frac{r_I}{g^*}\right)^{c-m}.$$

We call $\mathbf{M}_I$ and $\mathbf{M}'_I$ the "delay factors."

Now we prove that we may neglect Q'_I to obtain the base β of the principal exponential part of T.

We kept this term Q'_I because (17) could be useful to study the acceleration by considering solution nodes.

Dropping the factor $1-(g^*/g)^{c-s}$. We modify (16) by removing the term $-(g^*/g)^{c-s}$ so T has the new signification

$$T = \max_p 2^p \sum_{I,\mu,w} Q_{s,w}.$$

If we sum over w as above we finally obtain

$$T = \max_{\mu} \max_{m \le p} \sum_{|I|=m} Q_I. \tag{19}$$

where Q_I is still given by (18).

We will prove in two ways that our problem is reduced to finding $\lim_{v\to+\infty} T^{1/v}$ with these new expressions (18) and (19) for Q_I and T.

First proof. (This proof is valid for γ greater than the number γ_c we will define.) The new T is greater than the previous T because we removed $-(g^*/g)^{c-s}$. We will find that

$$\lim_{v\to+\infty} T^{1/v} = \beta = \max_{0\le p\le 1} 2^p \left(1 - \frac{3p^2}{8} + \frac{p^3}{4}\right)^{\gamma}.$$

It is easy to check that this maximum is reached for a value p_γ of p, with $p_\gamma < 1$ if $\gamma > \gamma_c$; this γ_c is easy to compute by considering the curves $2^p(1-(3p^2/8)+(p^3/4))^\gamma$. We have $4.5 \le \gamma_c \le 4.6$. For the value p_γ, g^*/g has value < 1, so $(g^*/g)^{c-s} < 1$, since $c-s \ge 1$. Thus, $\lim_{v\to+\infty} T^{1/v}$ is the same for the new T as for the previous T before $1-(g^*/g)^{c-s}$ is dropped. □

Second proof. We now suppose $\gamma > 1$. We will now show that we can drop the factor $\phi = (1-(g^*/g)^{c-w})^{1/v}$ in $T^{1/v}$ for $\gamma > 1$ without any additional restrictions.

We write $\phi = \left(1 - h(v,p)^{c-w}\right)^{1/v}$, where $h(v,p)$ is a rational expression of v, p, i.e., the quotient of two polynoms of degree 3 in v and p. We have exactly $h(v,v) = 1$ for every v.

From (16) we see that T is polynomially equivalent to the expression

$$T = \max_{p<v} \max_{m<p} \max_{w\le c-m} S(p,v,m,w)\left(1 - \left(\frac{g^*}{g}\right)^{c-m-w}\right).$$

We have $0 \le \frac{m}{v} \le \frac{p}{v} < 1$ and $\frac{w}{v} \le \gamma$. These quantities remain bound when v grows to infinity.

Thus, if $T^{1/v}$ has a limit β there exist functions p_1, m_1, w_1 of v such that p_1/v has a limit $\lambda_0 \le 1$, m_1/v and w_1/v have limits $l_1 < 1, l_2 < \gamma$, and

$$\beta = \lim_{v\to+\infty} T^{\frac{1}{v}}$$

$$= \lim\left[S(p_1(v), v, m_1(v), w_1(v))\right]^{\frac{1}{v}}\left[1 - \left[h(v, p_1(v))\right]^{c-m_1-w_1}\right]^{\frac{1}{v}},$$

$$h(v, p_1(v)) = \frac{g^*(p_1(v), v)}{g(p_1(v), v)}.$$

If $\lambda_0 = \lim_{v\to+\infty} p_1/v < 1$, $g^*(p_1(v), v)/g(p_1(v), v)$ has a limit < 1 (because $g^*/g \sim N/D \leq 1$ for $\lambda < 1$) and $\lim_{v\to+\infty} h(v, p_1(v)) = 0$, $\lim_{v\to+\infty} \phi = 1$.

Now we suppose $\lambda_0 = 1$.

Let $p = v(1-\eta(v))$, $\lim_{v\to+\infty} \eta(v) = 0$; since $p < v$, $\eta(v) \geq \frac{1}{v}$.

Now we know that $1 - (g^*/g)(p, v) \sim 2\eta/v + 2\eta^2$ (see §4.1), for $p = v(1-\eta)$. Thus,

$$s = 1 - \left[\frac{g^*}{g}(p, v)\right]^{c-w_1} = 1 - e^{(c-w_1)(\ln(1-2\eta/v-2\eta^2)+\text{higher-order terms})}$$

$$= 1 - e^{(\frac{c}{v}-\frac{w_1}{v})(-2\eta-2\eta^2 v+\text{higher-order terms})}.$$

Now two cases are possible.

1) $\lim_{v\to+\infty} \eta^2 v = 0$; thus,

$$s = 1 - \left[\frac{g^*}{g}(p, v)\right]^{c-w_1} = (\gamma - l_1)(2\eta/v + 2\eta^2) + \text{ higher-order terms.}$$

Thus $s^{1/v} = e^{(1/v)\ln\left((\gamma-l_1)(2\eta+2\eta^2+\text{higher-order terms})\right)}$ and $\lim_{v\to+\infty} s^{1/v} = 1$ because $2\eta + 2\eta^2 v \geq \frac{4}{v}$. (Roughly speaking $2\eta + 2\eta^2 v$ cannot decrease like the function e^{-v}, so $\lim_{v\to+\infty} s^{1/v}$ cannot be $\neq 1$.)

2) We suppose $\lim_{v\to+\infty} \eta^2 v = 0$ not true.

If $\lim_{v\to+\infty} s^{1/v} = 1$ is not true there exists a sequence of values of v for which $\lim_{v\to+\infty} \eta^2 v = l_3 > 0$ and for which $\lim_{v\to+\infty} s \neq 1$; then $\lim_{v\to+\infty} s^{1/v} = 1$. We have a contradiction. □

A third proof is the following.

Third proof. We have

$$\frac{g_p^*}{g_p} = 1 - \frac{4(1-\lambda+\frac{1}{v})(1-\lambda)(1-\lambda-\frac{1}{v}) + 6(\lambda-\frac{1}{v})(1-\lambda+\frac{1}{v})(1-\lambda)}{4(1-\frac{1}{v})(1-\frac{2}{v}) - 3(\lambda-\frac{1}{v})(\lambda-\frac{2}{v}) + 2\lambda(\lambda-\frac{1}{v})(\lambda-\frac{2}{v})} = 1 - y(\lambda).$$

The study of $y(\lambda)$ shows that it decreases on $]0, 1-\frac{1}{v}]$; so, $0 < 1 - y(\lambda) < 1 - y(1-\frac{1}{v}) = 1 - 4/(v^2 - v + 2)$, i.e., using $c - s \geq 1$,

$$0 < \left(\frac{g_p^*}{g_p}\right)^{c-s} \leq \left(1 - \frac{4}{v^2 - v + 2}\right)^{c-s} \leq \exp(-4/(v^2 - v + 2)).$$

Thus,

$$1 \geq \left[1 - \left(\frac{g_p^*}{g_p}\right)^{c-s}\right]^{\frac{1}{v}} \geq \left(1 - \exp(-4/(v^2 - v + 2))\right)^{\frac{1}{v}},$$

which has limit 1 when v grows to infinity; so we can replace $[1 - (g_p^*/g_p)]^{c-s}$ by 1.

Remarks on the average number of solutions. What follows will help the reader understand what happens when g^*/g is dropped and gives more information on our result (1) for $\gamma < \gamma_c$.

A clause which is not satisfied by $x_1, \ldots, x_v$ contains three elements of $\{-x_1, \ldots, -x_v\}$. The number of the clauses satisfied is $8\binom{v}{3} - \binom{v}{3} = 7\binom{v}{3}$; the number of satisfied entries is $(7\binom{v}{3})^{\gamma v}$. The number of all length v solution nodes, for all the entries, is $S = 2^v(7\binom{v}{3})^{\gamma v}$ and the number of cylinder solution nodes for every backtrack algorithm and in particular for **U** does not exceed $2S$.

Now, the average number S' of length v solution nodes is $(2(\frac{7}{8})^\gamma)^v$. We see that $S' = \beta^v$ for $\gamma < \gamma_c$. □

Remark. Define γ_0 such that $2\left(\frac{7}{8}\right)^{\gamma_0} = 1$; then $\gamma_0 = 5.19....$ For $\gamma > \gamma_0$ the average number S' has limit 0 when v increases; for $\gamma < \gamma_0$ it has limit ∞. $\gamma = \gamma_0$ is not possible because γ_0 is not rational, nevertheless it is possible to get sequences $(c_n, v_n)_{n\in N}$ for which c_n/v_n has limit γ_0 and such that the average number of solutions of the entries with v_n variables and c_n clauses with 3 literals remains in some fixed bounded interval.

Let $\gamma_m(v)$ be the ratio $\frac{c}{v}$ in the maximal entropy case (i.e., when half the entries are satisfied) (for $v = 50$, γ_m is close to 4.7). So $\gamma_m < \gamma_0$ and S' grows to infinity.

From now on we drop the factor $1 - (g^*/g)^{c-m-w}$ in the expression of T.

If we proceed with the elimination of w in the same way as above, we get

$$T = \max_p \max_{m<=p} \sum_{|I|=m} Q_I, \tag{20}$$

where

$$Q_I = 2^p \left(\frac{3}{4}\right)^m \frac{c!}{(c-m)!} \left(\frac{g}{(4v^3/3)}\right)^{c-m} \mathbf{M}_I \frac{1}{v^{3m}} c'_I \prod_i \frac{{u_i}^{t_i}}{t_i!},$$

$$\mathbf{M}_I = \left(1 + \frac{r_I}{g}\right)^{c-m}$$

As we did in §4.3, we now prove that we may replace g by $4v^3D/3$ in Q_I and $\mathbf{M}_I$.

We have to analyse

$$T = \max_p \max_{m<=p} \sum_{|I|=m} Q_I,$$

where

$$Q_I = 2^p \left(\frac{3}{4}\right)^m \frac{c!}{(c-m)!} D^{c-m} \mathbf{M}_I \frac{1}{v^{3m}} c'_I \prod_i \frac{{u_i}^{t_i}}{t_i!},$$

$$\mathbf{M}_I = \left(1 + \frac{3r_I}{4Dv^3}\right)^{c-m}, \tag{21}$$

$$D = 1 - 3\lambda^2/4 + \lambda^3/2, \qquad \lambda = \frac{p}{v}.$$

4.4. The k-partition on the set of the moderated measures. The average number of nodes is a sum of terms T_I which are indexed by the set $\mathbf{I}$ of the moderated I. The cardinality of this set $\mathbf{I}$ is not polynomial when $c\,v$ increases. To deal with this fact, we will introduce a partition on $\mathbf{I}$ with sets $\mathbf{I}_{m_1\ldots m_k}$, where the m_i are integers and $\sum_{1\le i\le k} m_i = m \le p \le v$. For given k the number of the sets $\mathbf{I}_{m_1\ldots m_k}$ is polynomial. Thus it only remains to find an $m_1 \ldots m_k$ for which $T_{m_1\ldots m_k} = \sum_I T_I$ is maximum, where I runs in $\mathbf{I}_{m_1\ldots m_k}$. Then we use variational arguments. We will let k grow infinitely. We will obtain a problem with real numbers, for which we use integration.

Let k be an integer. The segment $[0, 1[$ of real numbers is the union of the intervals $A_i = [\frac{i-1}{k}, \frac{i}{k}[$, where $i = 1, \ldots, k$. The sets vA_i form a partition of the segment $[1, v[$ of integers, we call this a k-partition.

Let $p < v$, $0 \le m \le p$, $0 \le m_i \le p$, $1 \le k$ be integers and $m = m_1 + m_2 + \cdots + m_k$ be a partition of m into k parts. We denote such a partition by the list $m_1 m_2 \ldots m_k$ and by $\mathbf{I}_{m_1\ldots m_k}$ denote the set of moderated measures $I = (t_i)_i$ on $[2, p-1]$ such that for $i = 1, \ldots, k$, $|I_i| = \sum_j t_j = m_i$, where j runs over $[(i-1)v/k, iv/k]$ and I_i denotes the restriction of I to the interval vA_i.

With a fixed k, T is polynomially equivalent to

$$T_k = \max_p \max_{m \le p} \max_{m_1 \dots m_k} (Q_{m_1 \dots m_k} - Q'_{m_1 \dots m_k}), \quad \text{where} \quad Q_{m_1 \dots m_k} = \sum_{I \in \mathbf{I}_{m_1 \dots m_k}} Q_I$$

and

$$Q'_{m_1 \dots m_k} = \sum_{I \in \mathbf{I}_{m_1 \dots m_k}} Q'_I.$$

This is true within a polynomial factor of degree $k+2$ in v.

Indeed, the number of the partitions $m_1 \dots m_k$ is less than v^k. We will take k large enough for our needs, but finite. If we find p and a partition of $m \le p$ such that $Q_{m_1 \dots m_k}$ is maximal, the exponential behaviour of T_k will be the same as that of $Q_{m_1 \dots m_k}$.

We now introduce the notation H, A_I, $A_{m_1 \dots m_k}$.

Let

$$(22) \quad Q_I = H A_I \quad \text{with} \quad A_I = \frac{1}{v^{3m}} \mathbf{M}_I c'_I \prod_i \frac{{u_i}^{t_i}}{t_i!}, \quad H = 2^p \left(\frac{3}{4}\right)^m \frac{c!}{(c-m)!} D^{c-m}.$$

So

$$(23) \qquad Q_{m_1 \dots m_k} = H A_{m_1 \dots m_k}, \quad \text{where} \quad A_{m_1 \dots m_k} = \sum_{I \in \mathbf{I}_{m_1 \dots m_k}} A_I.$$

4.5. An approximation for $A_{m_I \dots m_k}$. We will define approximations $A^+_{m_1 \dots m_k}$ and $A^-_{m_1 \dots m_k}$ of $A_{m_1 \dots k}$. First we define in §4.5.1 the maximal delay interval and study in §4.5.2 the delay terms r_I and $\mathbf{M}_I = (1 + (3r_I/4D))^{c-m}$.

4.5.1. Maximal delay intervals.

Definition of i_s. If there exists an integer j such that

$$(24) \qquad \sum_{j \le x \le i} m_x \ge (1 + i - j) \frac{v}{k}$$

for every $i \in [j, s]$, i.e., if the mass of the measure I is more than the number of positions in the stack between $j\frac{v}{k}$ and $s\frac{v}{k}$, then let i_s be defined as the least of these j; otherwise i_s is not defined.

Let i, j, s be integers in $[1, k]$. We say that $[i, j]$ is a k-m.d.i. (maximum delay interval as to $\mathbf{I}_{m_1 \dots m_k}$) if $i_j = i$ and $j = k$, or i_s is not defined for $s = j + 1$.

If $[i, j]$ is a k-m.d.i., $m_i \ge \frac{v}{k}$; if $i > 1$ then $m_{i-1} < \frac{v}{k}$; if $j < k$ then $m_{j+1} < \frac{v}{k}$. If $m_s \ge \frac{v}{k}$ then s is an element of an m.d.i.. If s is not in an m.d.i., then we say that s is without k-delay. The other s of $[1, k]$ are elements of k-m.d.i. $[i, j]$. If $m_s < \frac{v}{k}$, then s may be without delay or in an m.d.i. We denote by $M.D.I.$ the set of the k-m.d.i.

4.5.2. Approximation of the delay terms. We first approximate $r_I = \sum_i (i-1)\rho_i$ and $\mathbf{M}_I = \left(1 + (3r_I/4Dv^3)\right)^{c-m}$ for $I \in \mathbf{I}_{m_1 \dots m_k}$.

Let

$$(25) \qquad r^- = r^-_{m_1 \dots m_k} = \sum_{1 \le s \le k} \frac{v}{k}(s-1)\left(\tau_{s-1} - \frac{v}{k}\right)\frac{v}{k},$$

$$(26) \qquad r^+ = r^+_{m_1 \dots m_k} = \sum_{1 \le s \le k} \frac{v}{k} s \left(\tau_s + \frac{v}{k}\right)\frac{v}{k},$$

where τ_s is defined as follows.

Definition of τ_s.

$$\tau_s = \sum_{i_s \le x \le s} m_x - (1 + s - i_s)\frac{v}{k} \tag{27}$$

if i_s exists, otherwise by $\tau_s = 0$. (Without a loss of generality, we suppose that the (iv/k) are integers). We have for any $I \in \mathbf{I}_{m_1 \ldots m_k}$,

$$r^-_{m_1 \ldots m_k} \le r_I \le r^+_{m_1 \ldots m_k}.$$

We define $\mathbf{M}^- = \mathbf{M}^-_{m_1 \ldots m_k} = \left(1 + (3r^-/4Dv^3)\right)^{c-m}$ and $\mathbf{M}^+ = \mathbf{M}^+_{m_1 \ldots m_k} = \left(1 + (3r^+/4Dv^3)\right)^{c-m}$.

We may write

$$\mathbf{M}^-_{m_1 \ldots m_k} \le \mathbf{M}_I \le \mathbf{M}^+_{m_1 \ldots m_k}$$

for every $I \in \mathbf{I}_{m_1 \ldots m_k}$.

Remarks on i_s and τ_s. With these definitions of i_s and τ_s, $i_s = s$ is equivalent to $m_s \ge \frac{v}{k}$ and $m_{s-1} < \frac{v}{k}$, and $i_s = s$ and $m_s = \frac{v}{k}$ imply $\tau_s = 0$.

Moreover, $\tau_s = 0$ may happen in the two following cases:

1) i_s does not exist;

2) i_s exists, $i_s \le s$, and the excess $\sum_{i_s \le x \le s} m_x - (1 + s - i_s)\frac{v}{k}$ is zero.

4.5.3. Approximation of $A_{m_1 \ldots m_k}$. Now we recall that $c'_I = \prod_i (v - i + 1)$, where i runs over the set of the indices of the forced literals in n.

Definition of r_i. Let r_i be the index of the forcing literal of the forced literal l_i. For defining an upper bound $A^+_{m_1 \ldots m_k}$ of $A_{m_1 \ldots m_k}$ (see (23)) we will use the following modifications of

$$\sum_{I \in \mathbf{I}_{m_1 \ldots m_k}} c'_I \prod_{2 \le i \le p-2} \frac{{u_i}^{t_i}}{t_i!} = \sum_{I \in \mathbf{I}_{m_1 \ldots m_k}} \prod_{1 \le s \le k} \prod_{r_i \in vA_s} \frac{{u_{r_i}}^{t_{r_i}}}{t_{r_i}!}(v - i + 1).$$

1) If $s \in \{1, k]$ is without k-delay ($\tau_s = 0$) for any $I \in \mathbf{I}_{m_1 \ldots m_k}$ and for every index of forced literal i such that $r_i \in vA_s$ we replace the factors

$$\prod_{r_i \in vA_s} \frac{{u_{r_i}}^{t_{r_i}}}{t_{r_i}!}(v - i + 1)$$

by

$$\left(\frac{sv}{k}\right)^{m_s} \left(v - (s-1)\frac{v}{k} + 1\right)$$

to get an upper bound, and by

$$\left(\frac{(s-1)v}{k}\right)^{m_s} \left(v - (s+1)\frac{v}{k} + 1\right),$$

$$\left(\frac{(s-1)v}{k}\right)^{m_s} \left(v - (s+1)\frac{v}{k} + 1\right)$$

to get a lower bound.

For the s without delay we write $s \in$ W.D. The definition of r_i implies $i > r_i > (s-1)\frac{v}{k}$. Now, $\tau_s = 0$ implies that if a literal l_i is forced by l_j, with $j \leq s\frac{v}{k}$, then $i \leq (s+1)\frac{v}{k}$.

Now, replace $\sum_I \prod_i (1/t_{r_i}!)$ by $(1/m_s!)(\frac{v}{k})^{m_s}$. (Terms of the binomial formula may be lacking in $\sum(m_s!/\prod_{t_{r_i}}!)$ because of the moderation condition for I; decompositions $m_s = \sum_{j \in vA_s} t_j$ may be not available.

We can prove in three different ways that this modification has no effect on the rest of the analysis.

First proof. For given k, we will find in §4.8 integers $m \leq p < v$ and a distribution $m_1 \ldots m_k$ (with $m_k = 0$ for $k > \frac{p}{v}$) such that

$$Q_{m_1 \ldots m_k} = \sum_{I \in \mathbf{I}_{m_1 \ldots m_k}} Q_I$$

is maximum and $(Q_{m_1 \ldots m_k})^{1/v}$ has the limit β we are looking for.

When k is large enough, this distribution $m_1 \ldots m_k$ is close to a distribution with continuous density. This integral has no contact with the integral Δ of the constant distribution $m_s = v/k$; when k is large enough, we are sure that every I of $\mathbf{I}_{m_1 \ldots m_k}$ for an optimal $m_1 \ldots m_k$ is moderated. (By "has no contact" we mean that this integral is strictly less than the integral Δ; if equality happens we say that we have "contact.")

Second proof. A second proof could remain valid when changing the distribution of the data in such a way that we have a maximum $\mathbf{I}_{m_1 \ldots m_k}$ with contact. It consist of getting β from maximum distributions which are moderated in a weaker sense to avoid contact.

Third proof. A third proof, which could be used in more general situation, relies on the fact that even in the case when the term $\sum_I \prod_i (1/t_{r_i}!)$ is effectively smaller than $(1/m_s!)(\frac{v}{k})^{m_s}$, the neglected part is polynomialy equivalent to $(1/m_s!)(\frac{v}{k})^{m_s}$ [7, pp. 531, Ballot's problem].

2) The other s are distributed in the k-m.d.i. intervals $[i, j] \in M.D.I.$ We give an idea of what we will do. We also easily define for a single I its m.d.i. intervals in $[1, v]$. The restriction of I to such an m.d.i. $[x, y]$ flattens nearly over the same interval (shifted by 1) and the related factor in c'_I is $\prod_{z \in [x,y]}(v - z)$. When I varies in a $\mathbf{I}_{m_1 \ldots m_k}$ such that $[i, j]$ is a k-m.d.i., the related $[x, y]$ differ little from $[\frac{iv}{k}, \frac{jv}{k}]$ if k is large enough. Thus we will use the following approximation:

$$\prod_{[i,j] \in M.D.I.} \prod_{s \in [i,j]} \left(\frac{s\,v}{k}\right)^{m_s} \frac{1}{m_s!} \left(\frac{v}{k}\right)^{m_s} c^+_{i,j},$$

where $c^+_{i,j}$ is

$$c^+_{i,j} = \left(\frac{1}{v}\right)^{m_{i,j}} \prod_{v\frac{i-1}{k} \leq x \leq v\frac{j-1}{k} + \tau_j} (v - x + 1), \tag{28}$$

with $m_{i,j} = \sum_{s \in [i\,v, j\,v]} m_s$.

We get $A^+_{m_1 \ldots m_k}$

$$= \left(\frac{1}{v}\right)^{3m} \mathbf{M}^+ \prod_{l \in W.D.} \left(\frac{l\,v}{k}\left(v - \frac{l\,v}{k} + \frac{v}{k} + 1\right)\right)^{m_l} \frac{1}{m_l!} \left(\frac{v}{k}\right)^{m_l}$$
$$\prod_{[i,j] \in M.D.I.} \left(\prod_{i \leq s \leq j} \left(\frac{s\,v}{k}\right)^{m_s} \frac{1}{m_s!} \left(\frac{v}{k}\right)^{m_s}\right) c^+_{i,j}. \tag{29}$$

In a similar way

$$A^-_{m_1\dots m_k}$$

$$= \left(\frac{1}{v}\right)^{3m} \mathbf{M}^- \prod_{l\in W.D.} \left(\frac{(l-1)v}{k}\left(v - \frac{l\,v}{k} + 1\right)\right)^{m_l} \frac{1}{m_l!}\left(\frac{v}{k}\right)^{m_l} \prod_{[i,j]\in M.D.I.} \cdot \left(\prod_{i\le s\le j}\left(\frac{(s-1)v}{k}\right)^{m_s} \frac{1}{m_s!}\left(\frac{v}{k}\right)^{m_s}\right) c^-_{i,j}, \tag{30}$$

where $c^-_{i,j}$ is

$$c^-_{i,j} = \left(\frac{1}{v}\right)^{m_{i,j}} \prod_{v\frac{i}{k}\le x\le v\frac{j}{k}+\tau_j} (v - x + 1). \tag{31}$$

Let

$$Q^+_{m_1\dots m_k} = HA^+_{m_1\dots m_k},\ Q^-_{m_1\dots m_k} = HA^-_{m_1\dots m_k}. \tag{32}$$

A new use of the notations i, j, l, s. From now on $\frac{i}{k}, \frac{j}{k}, \frac{l}{k}, \frac{s}{k}$ are denoted by i, j, l, s ; so $i, j, l, s \in [0, 1]$ and the m.d.i. become intervals of $[0, 1]$. Also $p = \lambda v$ will change sense: $\lambda = p/v$ will be replaced by $p \le 1$. We will use the following convention: any index l is without delay (we write $l \in W.D.$); any index s is in an interval $[i, j]$ which is in $M.D.I.$

We set

$$\pi_l = l(1-l),$$

$$\pi^+_l = l(1-l) + \frac{1}{k} + \frac{1}{v}, \qquad \pi^-_l = \left(l - \frac{1}{k}\right)\left(1 - l + \frac{1}{v}\right). \tag{33}$$

So

$$A^+_{m_1\dots m_k} = \mathbf{M}^+\left(\frac{1}{k}\right)^m \left(\prod_l \frac{{\pi^+_l}^{m_l}}{m_l!} \prod_{[i,j]\in M.D.I.} \prod_s \frac{s^{m_s}}{m_s!}\right) \frac{c^+_{i,j}}{v^{(j-i)v}}, \tag{34}$$

and

$$A^-_{m_1\dots m_k} = \mathbf{M}^-\left(\frac{1}{k}\right)^m \left(\prod_l \frac{{\pi^-_l}^{m_l}}{m_l!} \prod_{[i,j]\in M.D.I.} \prod_s \frac{s^{m_s}}{m_s!}\right) \frac{c^-_{i,j}}{v^{(j-i)v}}. \tag{35}$$

4.6. The variational equations.

4.6.1. The variational inequations. From now on we do not mention the functions m_k and M_k which are defined at the end of the introduction. We have to deal with the fact that v tends to infinity and k has to be great enough. We prove that this approximation does not affect β, in the same way we did in §4.3 when replacing g by its equivalent.

For fixed $m \le p \le v$ we choose $m_1 \dots m_k$ for A^+ to be maximal. We use the following variational argument.

If we take off a unit mass from a $[l - \frac{1}{k}, l[$, A^+ is divided by π^+_l/m_l. If we add a unit mass to a $[l' - \frac{1}{k}, l'[$, A^+ is multiplied by $\pi^+_{l'}/(m_{l'} + 1)$.

So, for every l, l' without delay, the maximality condition is for every $l, l' \in W.D.$

$$\frac{\pi^+_l}{m_l} \le \frac{\pi^+_{l'}}{m_{l'}+1} \quad \text{and} \quad \frac{\pi^+_l}{m_l+1} \ge \frac{\pi^+_{l'}}{m_{l'}}. \tag{36}$$

Now, for fixed k and $\mu_1, \ldots, \mu_k$ real in $[0, 1]$ with $\mu = \mu_1 + \cdots + \mu_k$, we consider the $m_1 \ldots m_k$ defined by $m_i = \mu_i v$, which depend on one parameter v. So, $m = \mu v$.

We let v increase to infinity and we get the l—conditions

$$\frac{\mu_l}{\pi_l^+} = C', \tag{37}$$

where C' is a constant when l runs over $W.D.$

We consider now the s in an m.d.i. $[i, j]$.

If[1] we add a unit mass in $[s-1, s[$, a new forced literal is generated at the end of the m.d.i. So $c_{i,j}^+$ is multiplied by $1 - j'$ and the s-factor of A^+ is multiplied by $(1/(m_s + 1)s)(1 - j')$ with $j \le j' \le j + \frac{1}{k}$. The delay-factor $\mathbf{M}$ changes in the following way. The term $r_{m_1 \ldots m_k}$ increases by $\Delta r = v^2\big((j^2 - s^2)/2 + \epsilon\big)$ with $\epsilon = \frac{s+j}{2}\frac{1}{k}$ (later, using compacity, we let i and j be fixed and may neglect ϵ by taking k large enough).

$\mathbf{M}$ is multiplied by

$$x = \left(1 + \frac{\frac{j^2 - s^2}{2} + \epsilon}{\frac{4D}{3} + \frac{r}{v^3}}\right)^{c-m}.$$

Now we take $c = \gamma v$ with a fixed $\gamma > 1$.

With fixed k,

$$\lim_{v \to +\infty} x = \exp\Big((\gamma - \mu)(j^2 - s^2 + \epsilon)/2\left(\frac{4D}{3} + r\right)\Big) = \exp\big(\alpha(j^2 - s^2 + \epsilon)\big),$$

where now r is the following limit value:

$$r = \sum_{[i,j] \in M.D.I.} r_{i,j}, \tag{38}$$

where

$$r_{i,j} = \frac{(j-i)^2}{2}\left(\frac{j+i}{1 - e_j/e_i} - \frac{i+2j}{3}\right) - \frac{j-i}{\alpha}, \tag{39}$$

and

$$e_i = e^{-\alpha i^2}, \quad e_j = e^{-\alpha j^2}, \quad \alpha = \frac{1}{2}\frac{\gamma - \mu}{\frac{4D}{3} + r}, \quad D = D(p) = 1 - \frac{3p^2}{4} + \frac{p^3}{2}. \tag{40}$$

In the same way as above, we get the following conditions of maximality for $Q_{m_1 \ldots m_k}$. There is a constant C' such that for every l and every s,

$$\frac{\mu'}{\pi_l^+} = \frac{\mu_s}{s\, e^{-\alpha s^2}} \frac{1}{(1 - j')e^{\alpha j^2 + \epsilon}} = C'. \tag{41}$$

Replacing ϵ by 0 has no effect on the value of β in (1). This simplification does not contribute to the principal exponential part we are looking for.

(We say that these equations are the conditions for a global equilibration. In fact, for the calculation of the base we prefer to consider separately the equilibrations on the k-m.d.i. and on the set without delay, taking μ' below as a free parameter.)

[1] We see by the definition of τ_j that $\tau_j - (j - i + 1)\frac{v}{k} + m_{j+1} < \frac{v}{k}$ when I varies in $\mathbf{I}_{m_1 \ldots m_k}$. There is an interval of forced literals which always contains $v\left[\frac{i}{k}, \frac{j}{k}\right]$. The restriction of I to $v\left[\frac{i}{k}, \frac{j}{k}\right]$ flattens on an interval of length $(j - i + 1)\frac{v}{k} + \tau_j$. which has as an extreme position to the right $v\frac{i}{k}$, $v\frac{j}{k} + \tau_j$, where $\frac{1}{k} \le \tau_j \le \frac{2}{k}$, and as an extreme position to the left $v\frac{i-1}{k}$, $v\frac{j-1}{k} + \tau_j$.

4.6.2. The variational equations. We suppose that $j < p$. From now on we suppose that k is large enough to neglect here $\frac{1}{k}$. (It will be clearly possible when we will introduce integrals.) We consider the variational equations

$$\frac{\mu_l}{\pi_l} = \frac{e_j}{1-j}\frac{\mu_s}{s\,e^{-\alpha s^2}}, \text{ i.e., } \frac{\mu_s}{s\,e^{-\alpha s^2}} = \frac{C}{k}, \quad \frac{\mu_l}{\pi_l} = \frac{1}{a}\frac{C}{k}, \quad a = (1-j)/e_j. \tag{42}$$

The constant C is calculated with the condition that $\sum \mu_s = \mu_{i,j} = m_{i,j}/v$.

We now consider that s varies continuously on $[0, 1]$ with the Lebesgue measure $ds = \frac{1}{k}$, (when k increases enough). So we get limit values by integrals that may be easily neared with a finite k.

This approximation has no effect on the value of β in (1).

After elementary integration we get that

$$C = 2\alpha\frac{j-i}{e_i - e_j}. \tag{43}$$

4.7. Lemma "at most one maximal delay interval." Let $m_1 \ldots m_k$ be such that $Q_{m_1 \ldots m_k}$ is maximal; we say that $m_1 \ldots m_k$ is an equilibrated set of measures.

We will see that these equilibrated sets are of three types: 1) no delay–no constraint, 2) delay–no constraint, and 3) delay–constraint.

We will derive from a property of gaussian functions that there is at most one k-m.d.i., if $m_1 \ldots m_k$ is equilibrated. In the case without constraint it will reduce the search for an equilibrated $m_1 \ldots m_k$ to four real parameters γ, i, j, p and to three parameters γ, m, p in the case that there is no m.d.i. for $m_1 \ldots m_k$.

A first necessary condition for [*i*, *j*[to be an m.d.i. For $[i, j]$ to be a delay interval, we must have the following:

$$\text{for every } s \text{ with } i < s < j, \sum_{i \le \sigma \le s} \mu_\sigma \ge s - i. \tag{44}$$

As $\mu_\sigma = 2\alpha((j-i)/(e_i - e_j))\frac{1}{k}\sigma e^{-\alpha\sigma^2}$, with k infinitely large we get after elementary integration

$$\frac{e_i - e_s}{s-i} \ge \frac{e_i - e_j}{j-i}. \tag{45}$$

Let (G) be the gaussian curve $e^{-\alpha\sigma^2}$, $\sigma \in [0, 1]$. We say that condition (45) says that the "$i-s$ secant" is under the "$i-j$ secant," for every $s \in [i, j]$. The abscisse of the inflexion point of the gaussian curve is $\frac{1}{\sqrt{2\alpha}}$. So, if $j \ge i \ge \frac{1}{\sqrt{2\alpha}}$, (45) is true. If $j \le \frac{1}{\sqrt{2\alpha}}$, (45) is false.

Let $i_0 = i_0(\alpha, j)$ be the point of (G), the tangent of which passes by (j, e_j), for $j \ge \frac{1}{\sqrt{2\alpha}}$; then (45) is true if and only if $j \ge \frac{1}{\sqrt{2\alpha}}$ and $i \ge i_0(\alpha, j)$. The real number i_0 is easily calculated. We also derive from above the following lemma.

Lemma "at most one m.d.i. for an equilibrated $m_1 \ldots m_k$."

The parameters C and α are the same for all the m.d.i. of an equilibrated $m_1 \ldots m_k$.

As $m'_{i,j} = (e_i - e_j)/(j-i) = \frac{2k\alpha}{C}$, $m'_{i,j}$ is constant when $[i, j] \in M.D.I.$ By Rolle's lemma $m'_{i,j}$ is the absolute value of a tangent to (G) at a point over $]i, j[$. If $i > \frac{1}{\sqrt{2\alpha}}$ it is obvious that $m'_{i,j} > m'_{j,j}$, where $m'_{j,j}$ is the absolute value of the slope of the tangente in j. If there exists an m.d.i. $[i', j']$ left of $[i, j]$, $m'_{i,j} > m'_{i',j'}$, which cannot be true. We now only suppose that $j > \frac{1}{\sqrt{2\alpha}}$ and $i > i_0$. We use the convexity properties of (G). So, the "$i-j$ secant" has a slope larger than $m_{j,j}$ (in absolute value). We get the same contradiction if there were two m.d.i..

4.8. The three elementary functions f_i to be maximized. Now we will get the elementary functions f_i to be maximized. We finally obtain the simple formula in (1) for the average time of **UC**, as the consequence of the fact that only the first of these functions is relevant for the considered distribution.

4.8.1. Three classes for the sets of measures. We meet the following two difficulties which emerge from the fact that the t_i may be larger than 1, i.e., from the fact of flattening.

1) When, at node n, the reduced input F_n contains a surplus of forced literals, a so-called m.d.i. takes place. We overcome this fact with the so-called "at most one m.d.i." lemma.

2) The fact that the measures I must be "moderated" implies the possibility of constraint cases in the variational analysis. We still get the "at most one interval of constraint" property.

Thus, the $m_1 \dots m_k$ which may be equilibrated (we mean without constraint) are distributed in two classes.

The first one, $1.D.I$, is the class of the $m_1 \dots m_k$ for which there exists a m.d.i. $[i, j]$.

The second one, $0.D.I$, is the class of the $m_1 \dots m_k$ without any m.d.i. $[i, j]$.

We will study below the third case with constraint, which, roughly speaking, is necessary to keep a $m_1 \dots m_k$ from not satisfying the conditions of moderation.

4.8.2. Admissible cases (p, i, j), the first function $T_{p,i,j}$. The $m_1 \dots m_k$ which are equilibrated or partly equilibrated with constraint must satisfy the following conditions (k-mod) of k-moderation (we suppose without loss of generality that $p \in [\frac{1}{k}, \dots, 1]$):

$$m_x = 0 \quad \text{for } x > p,$$

$$\text{for every } s \in \left[\frac{1}{k}, p\right] : m_s + m_{s+\frac{1}{k}} + \cdots + m_p \leq (p - s + 1)v, \tag{46}$$

i.e., as $m_i = v\mu_i$,

$$\mu_s + \mu_{s+1/k} + \cdots + \mu_p \leq p - s \quad \text{for } k \text{ large enough.} \tag{k-mod}$$

In the $1.D.I.$ case, let $[i, j]$, $i, j \in [\frac{1}{k}, \dots, 1]$, be the k-m.d.i. for $m_1 \dots m_k$. The lemma "at most one m.d.i." implies the following second necessary condition:

$$\text{for every } l \notin [i, j],\ m_l \leq \frac{v}{k}, \quad \text{i.e., } \mu_l \leq \frac{1}{k}. \tag{47}$$

Conversely, the conjunction of (44) and (47) is a sufficient condition for $m_1 \dots m_k$ to be moderated (we recall that $\sum_{i \leq \sigma \leq j} \mu_\sigma = j - i$).

We say that (p, i, j), $0 \leq i \leq j \leq p \leq 1$, is *admissible* if the two conditions (44) and (47) are satisfied.

Now we use the variational conditions for μ_l to get the following expression for (47):

$$\text{for every } l \notin [i, j],\ \frac{a\, m'}{2\alpha} \geq \pi_l, \quad \text{where } m' = m'_{i,j} = (e_i - e_j)/(j - i). \tag{47'}$$

If $\frac{1}{2} \in [i, j]$, let $\sup(\pi_l) = \frac{1}{4}$, otherwise let $\sup(\pi_l) = \sup\{\pi_i, \pi_j, \pi_p\}$.

We now get the following expression for (47):

$$\text{for every } l \notin [i, j],\ \frac{a\, m'}{2\alpha} \geq \sup(\pi_l). \tag{48}$$

Now we will write the base of the principal exponential contribution $T_{p,i,j}$ of the set $m_1 \dots m_k$ of moderated measures in the $1.D.I.$ case, with an admissible (p, i, j). We put the

values of m_l, m_s from the variational equations into the expressions of $\lim_{v\to+\infty}(Q^{+}_{m_1\dots m_k})^{1/v}$ and $\lim_{v\to+\infty}(Q^{-}_{m_1\dots m_k})^{1/v}$.

In both cases we get expressions which are equivalent to the following $T_{p,i,j}$ when k is large.

$$T_{p,i,j} = \text{Debut}\left(\frac{S_{p,i,j}}{\mu'}\right)^{\mu'}\left(\frac{m}{2\alpha}e^{\alpha R'}\right)^{j-i}\frac{(1-i)^{(1-i)}}{(1-j)^{(1-j)}}\mathbf{M}, \tag{49}$$

where

(50)

$$\text{Debut} = 2^p D^{\gamma-\mu}\left(\frac{3}{4}\right)^{\mu}\frac{\gamma^{\gamma}}{(\gamma-\mu)^{(\gamma-\mu)}},\quad R' = \frac{i^2e_i - j^2e_j}{e_i - e_j},\quad \mu' = 2\frac{\alpha R}{a\,m'}S_{p,i,j},\quad R = \frac{1}{m},$$

$$m' = (e_i - e_j)/(j-i),\ S_{p,i,j} = \frac{p^2}{2} - \frac{p^3}{3} - \left(\frac{j^2}{2} - \frac{j^3}{3}\right) + \frac{i^2}{2} - \frac{i^3}{3},$$

$$D = D(p) = 1 - \frac{3p^2}{4} + \frac{p^3}{2},\quad e_i = e^{-\alpha i^2}, e_j = e^{-\alpha j^2},\quad \alpha = \frac{1}{2}\frac{\gamma-\mu}{\frac{4D}{3}+r},$$

$$\mathbf{M} = \left(1 + \frac{3r_I}{4Dv^3}\right)^{c-m},\qquad r = \frac{(j-i)^2}{2}\left(\frac{j+i}{1-e_j/e_i} - \frac{i+2j}{3}\right) - \frac{j-i}{\alpha}.$$

The $T_{p,i,j}$ may be calculated as follows.

At first we select only the admissible (p, i, j). When an admissible (p, i, j) is found, we calculate α as a fixed point, starting from $\alpha_0 = \frac{\gamma}{2}$. We always have the following properties:

$$r < \frac{1}{6},\qquad \alpha^- \le \alpha \le \alpha^+,$$

where

$$\alpha^+ = \frac{\gamma}{2}\quad\text{and}\quad \alpha^- = \frac{(\gamma-1)}{3}.$$

In fact, we will give in §4.9.2 an unexpected simplification. So we need not calculate the $T_{p,i,j}$. But it would be necessary to calculate the $T_{p,i,j}$ when changing the equidistribution of the data into another distribution which emphasizes the entries that result in delay.

4.8.3. The second function: $T_{p,\mu}$. Now we examine the $0.D.I$ case.

The parameters are $0 \le \mu \le p \le 1$. We get the following expression for $T_{p,\mu}$, the base of the principal exponential contribution of the corresponding set of measures $m_1 \dots m_k$:

$$T_{p,\mu} = \text{Debut}\left(\frac{S_p}{\mu}\right)^{\mu}, \tag{51}$$

where

$$\text{Debut} = 2^p D^{\gamma-\mu}\left(\frac{3}{4}\right)^{\mu}\frac{\gamma^{\gamma}}{(\gamma-\mu)^{(\gamma-\mu)}},\qquad S_p = \frac{p^2}{2} - \frac{p^3}{3}. \tag{52}$$

The $T_{p,\mu}$ must be considered only in the case when the (μ, p) are admissible in the following sense.

The variational equations reduce to $\frac{\mu_l}{\pi_l} = C$, for every $l \in \left[\frac{1}{k}, \dots, \frac{k}{k}\right]$. They imply

1) $\mu = C\,k\int_0^p \pi_t dt = C\,k\,S_p$, where $S_p = \frac{p^2}{2} - \frac{p^3}{3}$,

2) $\mu_l = \frac{1}{k}\frac{\mu}{S_p}\pi_l$.

We say that (μ, p) is admissible if the two following conditions are true.

First condition (mod.). The measure μ_l on $\left[\frac{1}{k}, \ldots, \frac{k}{k}\right]$ is moderated in the following sense:

$$\text{for every } i < p, \sum_{i \le l \le p} \mu_l \le p - i.$$

Second condition (no delay). For every l, $\mu_l < \frac{1}{k}$.

We get the following expression for the conditions (mod.) and (no delay):

$$\text{(Mod./53)} \qquad \text{If } p \ge \frac{1}{2} \text{ then } \mu \le A = \frac{8p^2}{3(2p+1)}, \text{ otherwise } \mu \le B = \frac{p\left(\frac{1}{2} - \frac{p}{3}\right)}{1-p}.$$

$$\text{(No delay/54)} \qquad \text{If } p \ge \frac{1}{2} \text{ then } \mu \le 4S_p, \text{ otherwise } \mu \le B.$$

4.8.4. The third function: $T_{p,\mu,i,c}$. We will see at the end of §4.9.2 that this function $T_{p,\mu,i,c}$ is not relevant. However, we give here some considerations about the "constraint" case to show that our method could deal with this case in other situations. We will now examine the variational case with constraints on a subset (C) of $P = \left[\frac{1}{k}, \frac{2}{k}, \ldots, \frac{p}{k}\right]$. Let x in P. We say that x is a contact for $m_1 \ldots m_k$ if $\sum_{s>x} m_s = p - x$. Let c be in P. We now only consider the $m_1 \ldots m_k$ which have c as least contact. We call them the $c \sim m_1 \ldots m_k$. We will rule out a subset of them which are not maximal. Let $T_{m_1 \ldots m_k}$ be the contribution of $m_1 \ldots m_k$ as an exponential base. Let a_i be the number by which $T_{m_1 \ldots m_k}$ is multiplied by adding one unit mass $\frac{1}{v}$ to the mass m_i.

We define the constraint set (C) for a maximal $T_{m_1 \ldots m_k}$ as the set of contact points; if $s \in (C)$, if $l < s$ then $a_s \ge a_l$; otherwise if $l > s$ then $a_s \le a_l$.

Lemma "at most one interval of constraint." For a maximal $c \sim m_1 \ldots m_k$, $(C) = [c, i]$ for an i in $[c, p]$. The proof considers a restriction of $m_1 \ldots m_k$ with two successive contacts c_1, c_2. We use a simple algorithm for equilibrating a $m_1 \ldots m_k$. There are two cases. If equilibrating fails by introducing a new contact, then we do recursion. If equilibrating succeeds, we use the fact that after equilibrating, the mass $m_{c_2} < \frac{1}{k}$. If $c_2 < p$ then for k large enough, we meet a contradiction with the constraint condition for $c_2 + \frac{1}{k}$ relative to $l = c_2$.

We also extend the lemma "at most one m.d.i." for ruling out delay from $[0, c]$.

Finally, the constraint case reduces to the following contribution:

$$\text{(55)} \qquad T_{p,\mu,i,c} = \text{Debut}\left(\frac{a\,m'}{2\alpha}\right)^{\mu'} \frac{i^i}{c^c} \frac{(1-c)^{(1-c)}}{(1-p)^{(1-p)}} \frac{1}{e^{2(i-c)}} e^{(j-i)\alpha R'} \left(\frac{m'}{2\alpha}\right)^{j-i} \mathbf{M}.$$

4.9. Expression of the principal exponential base of UC.

4.9.1. We now have enough to calculate the base of UC. Now we have reached our principal aim. Indeed, the calculation of the exponential base for **UC** for a γ consists of taking the maximum of the following three functions:

1) $T_{p,\mu}$ (this is the case of no delay interval); it is easy to see that

$$\text{(56)} \qquad \max_{0 \le \mu \le p} T_{p,\mu} = \max_p 2^p \left(1 - \frac{3p^2}{8} + \frac{p^3}{4}\right)^\gamma;$$

2) $T_{p,i,j}$ (for any admissible (p, i, j)).

Indeed this maximum is obtained for

$$\mu = \frac{3\gamma S_p}{4D + 3S_p}.$$

3) $T_{p,\mu,i,c}$ (this is the case of constraint). (We prove that c has only one possible value different from zero.)

These functions have no more than four real variables. Thus the exponential base may be calculated with any decimals with a microcomputer. This computing proved that the terms with delay $T_{p,i,j}$ and with constraint $T_{p,\mu,i,c}$ do not contribute to the maximum which is always reached by the term $T_{p,\mu}$ (case without delay and without constraint). In the special case of **UC** and of the chosen equiprobable distribution of the entries, we will give below another short proof without computing. (However, computing $T_{p,i,j}$, $T_{p,\mu,i,c}$ might be necessary for another probability on the set of the entries.)

4.9.2. A unexpected simplification: We prove theoretically that only the function $T_{p,\mu}$ is relevant. However, it happens that the theoretical optimisation of the $T_{p,i,j}$ and $T_{p,\mu,i,c}$ leads to a simple result: we prove now, by elementary analysis, that the maximum is reached in the case without delay and without constraint. This maximum is

$$T = \max_{0 \le p \le 1} 2^p \left(1 - \frac{3p^2}{8} + \frac{p^3}{4}\right)^\gamma. \tag{57}$$

Indeed this maximum is obtained for

$$\mu = \frac{3\gamma S_p}{4D + 3S_p}.$$

Proof. The term $T_{p,\mu,i,j}$ must be maximal when μ' varies, i, j and the m_s, $s \in [i, j]$, are constant. The condition $(\partial T_{p,\mu,i,j})/\partial\mu' = 0$ is equivalent to $(S_{p,i,j}/\mu') \times 3(\gamma - \mu)/(3D + 3r) = 1$, where $S_{p,i,j} = \int_0^i \pi_t\, dt + \int_i^p \pi_t\, dt$, $(\pi_t = t(1-t))$.

By matching the relations above for α and μ', we get $S_{p,i,j}/\mu' = \frac{1}{2\alpha}$ and $R \times e_j/1 - j = 1$, which may be written as

$$\frac{e_i}{1-i} = \frac{e_j}{1-j}. \tag{58}$$

By taking (58) into the conditions of equilibrium (41) for $m_1 \ldots m_k$ at i and $i-1$, we get

$$\frac{m_{i-1}}{\pi_{i-1}} \frac{1-i}{e_j} = \frac{m_i}{i \times e_i}, \quad \text{i.e.,} \quad \frac{m_{i-1}}{\pi_{i-1}} = \frac{m_i}{\pi_i}.$$

Because $m_{i-1} < 1$ and $m_i \ge 1$, by letting k grow to infinity, we get $m_i = 1$. So

$$\frac{S_{p,i,j}}{\mu'} = \frac{1}{\pi_i} \quad \text{and} \quad \pi_i = \frac{1}{2\alpha}. \tag{59}$$

Now taking (58) into the expression for

$$r = r_{i,j} = \frac{(j-i)^2}{2}\left(\frac{j+i}{1 - e_i/e_j} - \frac{i+2j}{3}\right) - \frac{j-i}{\alpha},$$

we get

$$r = \frac{t^2}{2} - \frac{t^2}{3}(2i + j),$$

where $t = j - i$. From the expression of 2α in (50) and from (59) we get

$$\frac{4D}{3} + r = (\gamma - \mu)\pi_i, \quad \text{where} \quad \mu = t + \mu' = t + \frac{S_{p,i,j}}{\pi_i}.$$

After straightforward simplifications we get

$$\frac{4}{3} - S_p = \gamma\pi_i, \tag{60}$$

where $S_p = (p^2/2) - (p^3/3)$; thus

$$T_{p,i,j} = 2^p\left(1 - \frac{3S_p}{4}\right)^{\gamma}. \tag{61}$$

For a given $\gamma \geq \frac{14}{3}$ (and p such that $S_p \geq \frac{4}{3} - \frac{\gamma}{4}$ if $\gamma \leq \frac{16}{3}$) we get the interval $[i, j]$ of maximality for $T_{p,i,j}$ as $[i_m, j_m]$, where i_m is a solution of (60) and j_m is a solution of (58) with $2\alpha = \gamma/(\frac{4}{3} - S_p)$. But the fact that the expression (61) is the same as in (57) allows us to conclude that the $T_{p,i,j}$ are not relevant.

The case of constraint is no more relevant; we majorize by removing the condition of moderation and get a $T_{p,i,j}$ term for which (61) is true.

4.9.3. Matching UC and B. The two lemmas of uniqueness of delay and constraint intervals have the following consequence: the base of the principal exponential part for given $\frac{c}{v}$ is the maximum of three functions of at most three real variables on a compact set.

The theoretical simplification §4.9.2 has the consequence that it is sufficient in this problem to compute the simple expression in (57).

We give the following example. For $\frac{c}{v} = 4$, the maximal contribution is given by a $T_{p,\mu} = 1.16$.

We now match with the asymptotic behaviour of the average number of nodes for Davis–Putnam without unit clause search **B**. It would be easier if we could find the base of **B**. For $\frac{c}{v} = 4$, the maximal contribution is 1.36. (For $\frac{c}{v}$ close to 1, the base in (57) is close to 2 because the algorithm **UC** is not good in this area.)

5. Remarks. We expect that our method will give the principal exponential time of some other variants of the Davis–Putnam procedure and of other algorithms. In our paper [9] we give such variants, the analysis of which is needed.

Our method could be extended to the case of an algorithm searching for one of the shortest clauses of the reduced entry F_n. For instance, such an algorithm works well for an entry with one clause.

The entries could also be supposed to have k-clauses, $k \geq 3$. In these cases, our method introduces measures other than I for each possible length of clause. But we think it would be easier to first find a simple proof that delay terms do not contribute to the principal exponential part of the average time. In the case of equal distribution of the entries this seems to us to be highly likely. In the present work we prove this fact only after finding the contribution of a maximum delay term $T_{p,i,j}$.

A pair P_1, P_2 of NP problems in the following situations could be matched with our method:

1) Let $\mathbf{A}_i$, $i = 1, 2$ be an algorithm which is known as the best one for P_i.

2) Let $D_{1,2}$ (resp., $D_{2,1}$), be a convenient data reduction from P_1 to P_2 (resp., P_2 to P_1), which satisfies a Levin condition of average case [8], [5]. Such a pair should be selected from more than 10^6 choices. But our "variational method" is unlikely to be applied in every

case. Clearly, applying this method may require several suitable conditions; for instance, it was possible for **UC** because, in this case, we had at our disposal a summation T which is explicit and without recursion, though not simple. Moreover, we could simplify the terms in this summation to get a simpler situation, then we could consider T as the sum of a polynomial number of parts $T_{m_1 \ldots m_k}$ of the summation T. Although these $T_{m_1 \ldots m_k}$ are sums of an exponential number of terms, it was possible in this case to consider them as close enough to each other, taking k large enough.

Acknowledgment. We thank Prof. Paul W. Purdom for his careful reading of this paper and his numerous helpful comments.

REFERENCES

[1] V. CHVATAL AND E. SZEMEREDI, *Many hard examples for Resolution*, J. Assoc. Comput. Mach., 35 (1988), pp. 759–768.

[2] M. DAVIS AND H. PUTNAM, *A computing procedure for quantification theory*, J. Assoc. Comput. Mach., 1 (1960), pp. 207–215.

[3] J. FRANCO, *Elimination of infrequent variables improves average case performance of satisfiability algorithms*, SIAM J. Comput., 20 (1991), pp. 1119–1127.

[4] A. GOLDBERG, P. PURDOM, AND C. BROWN, *Average time analyses of simplified Davis–Putnam procedures*, Inform. Process. Lett., 15, (1982), pp. 72–75.

[5] YURI GUREVICH, *Matrix decomposition problem is complete for the average case*, Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, MI, preprint.

[6] R. M. KARP, *The probabilistic analysis of some combinatorial search algorithms*, in J. F. Traub, ed., Algorithms and Complexity, Academic Press, New York, 1976.

[7] DONALD E. KNUTH, *The art of computer programming*, in Addison-Wesley, Vol. 3.

[8] LEONID A. LEVIN, *Average case complete problems*, SIAM J. Comput., 15, (1986), pp. 285–286.

[9] H.-M. MÉJEAN, H. MOREL, G. REYNAUD, *Some remarks about SL-resolution and Davis–Putnam procedures*, Tech. Rep. 2, 1989, Lab. A.P.I., Fac. Sci. Luminy, case 901, 13288 Marseille, Cedex, France.

[10] MING-TE CHAO AND J. FRANCO, *Probabilistic analysis of two heuristics for the 3-Sat. problem*, SIAM J. Comput., 15 (1986), pp. 1106–1118.

[11] B. MONIEN AND E. SPECKENMEYER, *Solving satisfiability in less than 2n steps*, Discrete Appl. Math. 10 (1985), pp. 287–295.

[12] P. W. PURDOM, JR, *Search rearrangement backtracking and polynomial average time*, Artif. Intell., 21 (1983), pp. 117–133.

[13] P. W. PURDOM AND C. A. BROWN, *An analysis of backtracking with search rearrangement*, SIAM J. Comput., 12 (1983), pp. 717–733.

[14] P. W. PURDOM, JR, CYNTHIA A. BROWN, AND E. L. ROBERTSON, *Backtracking with multilevel dynamic search rearrangement*, in Acta Inform., 15, pp. 99–113.

[15] P. W. PURDOM, JR, *A survey of average time analyses of satisfiability algorithms*, J. Inform. Process., 13 (1990), pp. 449–455.

SIAM J. COMPUT.
Vol. 24, No. 3, pp. 650–664, June 1995

014

A GENERAL APPROACH TO REMOVING DEGENERACIES*

IOANNIS Z. EMIRIS† AND JOHN F. CANNY†

Abstract. We wish to increase the power of an arbitrary algorithm designed for nondegenerate input by allowing it to execute on all inputs. We concentrate on infinitesimal symbolic perturbations that do not affect the output for inputs in general position. Otherwise, if the problem mapping is continuous, the input and output space topology are at least as coarse as the real euclidean one, and the output space is connected, then our perturbations make the algorithm produce an output arbitrarily close or identical to the correct one. For a special class of algorithms, which includes several important algorithms in computational geometry, we describe a deterministic method that requires no symbolic computation. Ignoring polylogarithmic factors, this method increases the worst-case bit complexity only by a multiplicative factor which is linear in the dimension of the geometric space. For general algorithms, a randomized scheme with an arbitrarily high probability of success is proposed; the bit complexity is then bounded by a small-degree polynomial in the original worst-case complexity. In addition to being simpler than previous ones, these are the first efficient perturbation methods.

Key words. input degeneracy, ill-conditioned problems, symbolic perturbation, infinitesimals, randomization, determinants, roots of polynomials, algorithmic complexity

AMS subject classifications. 68Q10, 68Q20, 68Q25, 68U05

1. Introduction.

Quite often algorithms are designed under the assumption of input nondegeneracy. Although they can have many specific forms, most degeneracies in geometric or algebraic algorithms reduce to a division by zero, or to a sign determination for a value which is zero. In this article we describe efficient methods for systematically removing such degeneracies using symbolic infinitesimal perturbations. Our methods apply to every algorithm that can be implemented on a real random access machine (RAM).

This work is influenced by the treatment of the problem in [12] and, in a more general context, [21]. The main contribution of this article is to introduce the first general and efficient perturbations from the viewpoint of worst-case complexity. Previous methods incurred an extra computational cost that was exponential in some parameter of the input size.

The principal domains of applicability are geometric and algebraic algorithms over an infinite ordered field. Take, for instance, a convex hull algorithm in arbitrary dimension over the reals. It is typically described under the hypothesis of general position which excludes several possible instances, such as more than k points lying on the same $(k-1)$-dimensional hyperplane, in m-dimensional euclidean space, $m \geq k$. For an algebraic algorithm, consider Gaussian elimination without pivoting that works under the hypothesis that the pivot never vanishes. Our perturbation scheme accepts a program written under this hypothesis and outputs a slightly longer program that works for all inputs.

The perturbations introduced change the original input instance into a nondegenerate one which is arbitrarily close, in the usual euclidean metric, to the original input. For algorithms that branch only on the sign of determinants, which includes several important geometric algorithms, we propose a deterministic method. It increases the worst-case algebraic complexity of the algorithm by a multiplicative factor of $O(\log d)$ and its worst-case bit complexity by a factor of $O(d^{1+\alpha})$, where d is the dimension of the geometric space of the input objects and α is an arbitrarily small positive constant accounting for the polylogarithmic factor. In addition to its efficiency, this scheme is easy to implement, which makes it attractive for practical

*Received by the editors August 17, 1992; accepted for publication (in revised form) January 21, 1994. This work was supported by a David and Lucile Packard Foundation Fellowship and by National Science Foundation Presidential Young Investigator grant IRI-8958577. A preliminary version appeared in the *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, 1991.

†Computer Science Division, University of California at Berkeley, Berkeley, California 94720.

use [14]. The perturbation, although defined in terms of a symbolic infinitesimal variable, does not require any symbolic computation.

For general algorithms, we propose a randomized scheme that incurs a factor of $O(D^{1+\alpha})$ on the algebraic complexity, where D is the highest total degree in the input variables of any polynomial in the program. Under the bit model, the worst-case running time of the new program is asymptotically bounded by $\phi^{3+\alpha}$, where ϕ is the original bit complexity of the original one; in both cases α denotes an arbitrarily small positive constant.

All claims about the efficiency of our approach are based on worst-case complexities. Yet there exist other measures, such as output size, under which the perturbations cause a much more significant increase in running time. For a degenerate input the output may be of constant size while, under the perturbation, the given algorithm cannot do significantly better than what its worst-case performance predicts.

The next section defines the computational model, formalizes the use of infinitesimals as well as the notion of degeneracy, and specifies the problem at hand. Section 3 is a comparative study of previous work on handling degeneracies. Sections 4 and 5 describe the perturbations for algorithms that branch on determinants and on arbitrary rational functions, respectively; each section includes an application. We conclude with a summary and a discussion of directions for further work.

2. Preliminaries.

2.1. Model of computation. Our results hold for any infinite ordered field, yet we present them in terms of the reals $\mathbb{R}$. We choose the real-arithmetic RAM as our model; it is described in [18] and is a more powerful version of the simple RAM defined in [1].

An *input* of size N consists of a finite real vector $\mathbf{x} = (x_1, \ldots, x_N)$ and a particular input instance is $\mathbf{a} = (a_1, \ldots, a_N) \in \mathbb{R}^N$. The real RAM can perform real arithmetic exactly with respect to the four basic operations in $\{+, -, *, /\}$ and can branch on the sign of a rational function in the input variables, evaluated at the particular input instance. The machine can also write to and read from a memory that can store an arbitrary number of exact real values. The chosen model abstracts certain issues that may arise in practice, such as real number representation and exactness of arithmetic operations.

The set of instructions that implements a given algorithm on the real RAM forms a *program*; no program can alter itself. The subset of instructions executed on some input instance forms an *execution path*. For a specific program on the real RAM and a given input, the *output* is unique and is expressed by a finite real vector, possibly describing other structures such as graphs.

In §5 we shall require an extension of the model, namely that there exists an explicit finite integer bound D on the total degree in $\mathbf{x}$ of any polynomial computed in the course of the program. The concept of this bound appears in the machine of [2] and in the algebraic decision tree of [18].

Under the *algebraic model*, the worst-case complexity of an algorithm equals the maximum number of arithmetic operations, branching, and memory access instructions executed on any input. More realistically, we may wish to consider the effect of the operands' bit size on the speed of arithmetic operations. Under the *bit model* there is a cost function on each instruction of the program and the worst-case complexity of an algorithm equals the maximum sum of the costs of all instructions on the execution path corresponding to any input. In both models, the time to access the memory is assumed constant and therefore does not affect the total asymptotic running time. Branches also take constant time, provided that each number carries an extra bit indicating whether it equals zero or not. Without this extra bit the branching cost would be linear in the size of the operand but this would not affect the results in this article.

The cost of arithmetic operations depends on the particular operation executed as well as the bit size of the operands. For integers of size b, addition and subtraction have cost $O(b)$, while the cost of multiplication and division, due to an algorithm by Schönhage and Strassen, is $O(b \log b \log\log b)$ [1]. For rationals, the greatest common divisor (GCD) is factored out at every arithmetic operation, and finding it takes $O(b \log^2 b \log\log b)$ time [1]. Let $M(b) = O(b \log^2 b \log\log b)$ bound the bit complexity of any operation on two rational numbers, each represented by a pair of $O(b)$-bit integers. We define the bit size of a rational number to be the maximum bit size of the numerator and denominator.

Given an input instance, our perturbations define a new instance in terms of a symbolic variable that is never evaluated, which implies that instead of real numbers the program may have to manipulate polynomials in this variable. Formally, this can be thought of as producing a new program with the same control flow in which every arithmetic and branching instruction is substituted by a black box that implements the appropriate operation on univariate polynomials. Of course, the algebraic as well as the bit cost associated with each instruction changes.

2.2. Infinitesimals. Our approach in removing degeneracies is to add to the input values arbitrarily small quantities. To this effect we make use of infinitesimals. The process of extending the field of reals by an infinitesimal is a classical technique, formalized in [3] and used by the second author in [4].

DEFINITION 2.1. *We call* ϵ infinitesimal *with respect to* $\mathbb{R}$ *if the extension* $\mathbb{R}(\epsilon)$ *is ordered so that* ϵ *is positive but smaller than any positive element of* $\mathbb{R}$.

Clearly, the sign of any polynomial in ϵ is the sign of the nonzero term of lowest degree.

Alternatively, it is enough for ϵ to belong to the reals and take a sufficiently small positive value so that it avoids the roots of a finite set of polynomials; we shall see that this is the set of all polynomials appearing at a branch in the real RAM program. The smallest positive root in any of these zero sets is larger than some positive real ϵ_0, hence it suffices that $\epsilon = \epsilon_0$. This idea may be seen as a special case of the "transfer principle" [20].

An immediate consequence is that symbolic perturbations of the input by ϵ-polynomials are equivalent to defining a new real instance by setting ϵ equal to ϵ_0. Then the execution path on perturbed input is that of some real input which implies that the algorithm halts on perturbed input provided that it does for all real inputs.

2.3. Degeneracy. Before formalizing the notion of degeneracy, we examine it with respect to some concrete problems. For the matrix inversion problem an intrinsically degenerate input is a singular matrix, for which the output is undefined. An input degeneracy may depend not only on the particular problem but also on the algorithm. An algorithm-induced degeneracy for the Gaussian elimination algorithm without pivoting arises at a matrix with a singular principal minor.

Yap in [21] uses the convex hull problem in the plane to distinguish between intrinsic and algorithm-induced degeneracies. Assume that in the output space topology polytopes of distinct combinatorial structure lie in disjoint components. Then three collinear points constitute an intrinsic degeneracy because the mapping of point sets to convex hulls is not continuous. On the other hand, two covertical points have nothing special with respect to the mapping of point sets to their convex hull. They may, however, constitute a degeneracy with respect to a particular algorithm that solves the problem by using a vertical sweep-line or relies on some vertical partitioning of the plane.

We formalize the discussion by considering both input and output spaces as real topological subspaces of finite dimension.

DEFINITION 2.2. *A* problem mapping *associates with almost every input instance exactly one (exact) solution.*

DEFINITION 2.3. *The input instances on which the problem mapping is not defined or not continuous form the set of* intrinsic *degeneracies for this problem.*

An algorithm and, equivalently, the respective real RAM program that compute a problem mapping typically impose certain restrictions on the input instances. Hence the need to consider the mapping defined by a specific algorithm.

DEFINITION 2.4. *An* algorithm mapping *is defined by a particular real RAM program and is a restriction of the problem mapping to exclude at least all intrinsic degeneracies.*

In what follows no distinction is made between an algorithm and the real RAM program that implements it.

DEFINITION 2.5. *The* input space *of an algorithm mapping is a real space of finite dimension. In the context of the computational model, defined in* §2.1, *the dimension equals N. The* output space *of an algorithm mapping is a topological space, equal to the union of the disjoint finite-dimensional real topological spaces associated with the distinct execution paths of the real RAM program. These output subspaces will be called* leaf subspaces.

This terminology reflects the fact that branching causes the program to have a tree structure. The problem and algorithm output spaces can be either connected or disconnected. Usually, a disconnected output space can be made connected by identifying points in different leaf subspaces. Then the overall space inherits the topology of the leaf subspaces with no open sets intersecting two leaf subspaces.

This is possible in the example of the convex hull problem mapping, where polytopes which are identical as point sets but lie in different subspaces can be identified, thus producing a connected output space. Yet the problem mapping remains discontinuous. There exist other topologies that will make this space connected and the problem mapping continuous. One example is the metric topology where the distance of two polytopes is measured by the volume of their symmetric difference.

DEFINITION 2.6. *An input instance is* degenerate *with respect to some algorithm if, during its execution, it causes some branch rational function f, whose numerator and denominator polynomials are not identically zero, either to be undefined or to evaluate to zero while the algorithm produces no solution for the case $f = 0$. Equivalently, the input instance is in* general position *or* generic *if there is no such test function f.*

Clearly, the domain of an algorithm mapping is precisely the set of all generic inputs, which excludes *all* degeneracies, i.e., both intrinsic and induced ones. An important class of degenerate inputs are those that lead a program to division by zero. This case is included in the previous definition by requiring that the program be robust enough to have a zero test on the denominator before each division.

DEFINITION 2.7. *The set of* induced degeneracies *includes exactly those degenerate inputs that are not intrinsic degeneracies.*

The input space can be partitioned into equivalent classes, where each instance produces the same sign sequence on the branch rational functions reached during execution. The classes that do not make any branch function f, as specified in Definition 2.6, vanish or be undefined, partition the domain of the algorithm mapping into cells of input instances that produce an output instance in the same leaf subspace. The union of inputs that cause some branch polynomial to vanish contains the degenerate subset and has positive codimension since it is the finite union of polynomial zero sets. Hence the degenerate subset has positive codimension which agrees with the informal view of degeneracies as special cases or events of zero probability.

2.4. Problem definition. Consider an algorithm that solves a problem under the hypothesis of nondegeneracy. Our aim is, given an arbitrary input instance $\mathbf{a} = (a_1, \ldots, a_N)$, to define in a systematic way some other instance so that the same algorithm can always produce

a meaningful output. The new instance, denoted $\mathbf{a}(\epsilon) = (a_1(\epsilon), \ldots, a_N(\epsilon))$, will be defined by adding to each a_i a polynomial in some symbolic positive infinitesimal ϵ. The discussion in §§2.1 and 2.2 implies that $\mathbf{a}(\epsilon)$ can be given as input to the same real RAM program.

To provide some intuition on the desired effects of perturbations we return first to the matrix inversion problem. Given a perturbed singular matrix, (i) the algorithm should return its inverse and (ii) the perturbed input should be arbitrarily close to the original one under the standard euclidean metric. Condition (i) will follow from the correctness of the algorithm on generic inputs once we establish that perturbed matrices are nonsingular. Restricted to nonsingular matrices, the problem mapping is continuous. On a perturbed nonsingular matrix, (ii) implies that the output is arbitrarily close to the exact solution; to recover the latter we can simply set the infinitesimal to zero in the perturbed output.

There is always a solution for the convex hull problem; however, its combinatorial nature may prohibit the problem mapping from being continuous. Take, for instance, the volume of the symmetric difference between the actual output and the exact convex hull as the distance between these two polytopes. This volume tends to zero with ϵ since the induced metric topology is at least as coarse as the euclidean one. Under this metric the output space is connected and the approximate solution is arbitrarily close to the exact one.

We now specify the desired properties of a perturbation in terms of the outputs obtained under different circumstances. Limits are understood with respect to the topology of the input and output spaces.

DEFINITION 2.8. *Given an input instance* $\mathbf{a}$, *a* strongly valid perturbation *defines a new instance* $\mathbf{a}(\epsilon)$ *which lies in general position, tends to* $\mathbf{a}$ *as* ϵ *approaches zero, and satisfies the following conditions:*

1. *If* $\mathbf{a}$ *is in general position, the algorithm produces the same output whether it runs on* $\mathbf{a}$ *or it runs on* $\mathbf{a}(\epsilon)$ *and at the end* ϵ *is set to* 0.

2. *If* $\mathbf{a}$ *is an induced degeneracy, the output space is connected with topology not finer than the euclidean one, and the problem mapping is continuous, then the algorithm on* $\mathbf{a}(\epsilon)$ *returns an output that either produces the exact solution by setting* $\epsilon = 0$ *or tends to the exact solution in the limit as* $\epsilon \to 0$.

3. *If* $\mathbf{a}$ *is degenerate and some hypothesis of the previous case fails, then the algorithm produces a correct solution for* $\mathbf{a}(\epsilon)$.

A more practical definition describes requirements for a weaker perturbation in terms of the input space.

DEFINITION 2.9. *Given an input instance* $\mathbf{a}$, *a* valid perturbation *defines a nondegenerate instance* $\mathbf{a}(\epsilon)$ *which tends to* $\mathbf{a}$ *as* ϵ *approaches zero, such that when* $\mathbf{a}$ *is in general position all branches take the same direction on* $\mathbf{a}(\epsilon)$ *as on* $\mathbf{a}$.

PROPOSITION 2.10. *Suppose that the input space and leaf subspace topologies are at least as coarse as the real euclidean topology. Then any valid perturbation is strongly valid.*

Proof. It suffices to show that the three conditions of Definition 2.8 are satisfied. For generic inputs all branches take the same direction on $\mathbf{a}$ and $\mathbf{a}(\epsilon)$, hence the two outputs lie in the same leaf subspace. No ϵ-term can be the most significant in any polynomial in the output, because this is the output on a generic instance. Therefore all such terms can be ignored by setting $\epsilon = 0$, thus obtaining the exact solution.

In the second case, since we deal with an induced degeneracy, an exact solution exists. Whenever the problem mapping is continuous the algorithm mapping is also continuous in its own domain. Since the input space topology is sufficiently coarse, the fact that $\mathbf{a}(\epsilon)$ tends to $\mathbf{a}$ and the continuity property imply that the output produced on perturbed input tends to the exact solution as $\epsilon \to 0$. In the more favorable cases, the exact output is recovered by setting $\epsilon = 0$.

The correctness of the algorithm on generic inputs and the hypothesis that $\mathbf{a}(\epsilon)$ is generic imply that the last condition is satisfied. □

In what follows we focus on problems that satisfy the hypothesis of the previous proposition, hence reducing the validity requirements to those of Definition 2.9. Furthermore, we consider perturbations of the following form:

$$a_i(\epsilon) = a_i + \epsilon\, c_i,$$

for $c_i \in \mathbb{Z}$ independent of a_i and ϵ. The postprocessing necessary to recover the exact answer is usually a very case-specific process. We discuss the case of convex hulls at the end of §4 and also refer the reader to [21], [12], [8].

3. Other work. The most naive approach is to handle each special case separately, which is tedious for implementors and unattractive for theoreticians. Random perturbations are frequently alluded to and one such scheme is studied in this article. Their main feature is that they trade randomness for efficiency.

Symmetry breaking rules in linear programming are the earliest systematic approaches to the problem. Dantzig presents such a method in [6] which relies on an infinitesimal ϵ. Consider a linear program reduced to finding nonnegative values for the $m + n$ variables x_j, such that the sum of all slack variables $\sum_{j>n}^{m+n} x_j$ is minimized. The perturbation consists in adding a power of ϵ to every nonnegative constant b_i, where $1 \le i \le m$:

$$\sum_{j=1}^{n} a_{i,j}x_j + x_{n+i} = b_i + \epsilon^i.$$

This forces the perturbed constants to be strictly positive and eliminates the degenerate case of having $b_i = 0$, for some i in $\{1, \ldots, m\}$.

Edelsbrunner and Mücke systematize in [12] a scheme called simulation of simplicity (SoS for short) already presented in [9], [11], [13], [10]. It applies to algorithms that accept n input objects, each specified by d parameters, and whose tests are determinants in the nd parameters, just as our deterministic perturbation (1) of the next section. SoS perturbs every input parameter $p_{i,j}$ into

$$p_{i,j}(\epsilon) = p_{i,j} + \epsilon^{2^{i\delta - j}}$$

where $\delta > d$ and ϵ is a symbolic infinitesimal; this is a valid scheme under our definition. The sign of the perturbed determinant is the sign of the smallest-degree term in its ϵ-expression and can be calculated numerically.

Finding the sign of the perturbed determinant is, on the average, fairly fast. In the worst case, however, the determinant computation takes $\Omega(2^d)$ steps, since it may have to check that many minors of the perturbed matrix. This bound is obtained by calculating the number of distinct vectors $(v_1, \ldots, v_{d-2})$, where d denotes the order of the original matrix. Every v_i is a positive integer less than or equal to d and for every $i < j$, $v_i \le v_j$. In [12] every such vector is associated with a distinct minor that may have to be evaluated. This analysis pertains to Λ matrices, to be defined in the next section. Matrices of the second kind, the Δ matrices, require more steps in the worst case, for the same order. In short, SoS incurs a worst-case exponential overhead in d.

Yap in [21] provides a more general framework, which includes SoS as a special case, where branching occurs at arbitrary rational functions. His technique is consistent relative to infinitesimal perturbations [22] and valid; here we examine it as applied to polynomial tests. Let $PP = PP(x_1, \ldots, x_n)$ denote the set of all power products of the form

$$w = \prod_{i=1}^{n} x_i^{e_i}, \quad e_i \geq 0$$

in the n input variables. A total ordering $\leq_A$ on PP is *admissible* if, for all $w, w', w'' \in PP$,

$$1 \leq_A w \qquad \text{and} \qquad w \leq_A w' \Rightarrow ww'' \leq_A w'w''.$$

Let $w_1, w_2, \ldots$ be the ordered list of power products larger than 1, i.e., those with at least one positive exponent. Then each polynomial $f(\mathbf{x})$ is associated with the infinite list

$$S(f) = (f, f_{w_1}, f_{w_2}, \ldots),$$

where f_{w_k} is the partial derivative of f with respect to w_k; for example, $f_{x_2^2 x_3} = \partial^3 f/(\partial^2 x_2\, \partial x_3)$. The sign of a nonzero polynomial f is the sign of the first polynomial in $S(f)$ whose value at the actual input is not zero, which can always be found after examining a finite number of terms.

Yap focuses on sparse n-variate polynomials, with m denoting the maximum degree of any variable. In the case when all variables are of degree m, a polynomial f has $(m+1)^n - 1 \geq m^n$ nontrivial derivatives. On the average only a few partial derivatives will have to be evaluated, but at worst all of them have to be computed and the complexity is $\Omega(m^n)$.

Dobrindt, Mehlhorn, and Yvinec [8] studied the problem of intersecting an arbitrary polytope with a convex one in three dimensions, proposed an efficient perturbation, and discussed postprocessing in this context. The interesting feature of their technique is that it controls the direction of perturbation. In particular, since the facet structure is given, the polytope vertices are forced to be perturbed outward.

In a slightly different vein, Canny used a *structural* perturbation in [4] to ensure that the input semialgebraic sets are in general position. One immediate application is to motion-planning algorithms, where these sets describe obstacles or prohibited space. The perturbation preserves the emptiness and number of connected components of the original sets using sequences or towers of infinitesimals.

Perturbation methods have been applied in other cases to eliminate degeneracies with respect to particular problems, as in [17], for instance.

Finally, Emiris and Canny in [14] extend the applicability of the deterministic perturbation introduced in this article to another two geometric branching tests, most importantly to the InSphere test. They also propose a new variant of the scheme that eliminates the polynomial factor in the asymptotic bit complexity overhead with respect to the two tests examined here.

4. Branching on determinants. We first restrict attention to algorithms whose branching depends exclusively on the sign of determinants in the input variables, which is the case with several geometric algorithms. We concentrate on two specific types of determinants that cover important algorithms, such as those computing convex hulls and hyperplane arrangements. Our approach can be applied to other types of determinants too, as demonstrated in [14].

Assume that the input parameters represent n input objects $p_1, p_2, \ldots, p_n$, each specified by d parameters. Without loss of generality, each $p_i = (p_{i,1}, p_{i,2}, \ldots, p_{i,d})$, is a point in $\mathbb{R}^d$. We are interested in the case that the dimension d is arbitrary, whence the total input size is nd. It is also reasonable to assume that a constant fraction of the points are distinct, thus establishing a lower bound on the parameters' bit size. In practice this condition can be guaranteed using an initial check to eliminate duplicate points; in most settings the complexity of this phase is dominated by that of the main algorithm.

We perturb deterministically every parameter $p_{i,j}$ to obtain $p_{i,j}(\epsilon)$, where ϵ is an infinitesimal symbolic variable:

$$p_{i,j}(\epsilon) = p_{i,j} + \epsilon\,(i^j). \tag{1}$$

Although ϵ is never assigned a real value, we shall show that no symbolic computation is necessary.

First consider matrix Λ_{d+1} whose rows correspond to points $p_{i_1}, p_{i_2}, \ldots, p_{i_{d+1}}$:

$$\Lambda_{d+1} = \begin{bmatrix} 1 & p_{i_1,1} & p_{i_1,2} & \cdots & p_{i_1,d} \\ 1 & p_{i_2,1} & p_{i_2,2} & \cdots & p_{i_2,d} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & p_{i_{d+1},1} & p_{i_{d+1},2} & \cdots & p_{i_{d+1},d} \end{bmatrix}.$$

Testing the sign of this determinant comes up in various contexts. We call it the sidedness test because, given a query point $p_{i_{d+1}}$ and the hyperplane spanned by the other d points, the sign of the determinant indicates on which side of the hyperplane the query point lies. The determinant vanishes if and only if the query point lies on the hyperplane; the positive and negative sides of the hyperplane are determined by the order of the d points defining it. This test is sometimes called the orientation test, since it may be regarded as deciding the relative orientation of the $d + 1$ points in the sense of [12]. In fact, the column of ones should be rightmost, but it is a constant-time operation to obtain the orientation of the points from the sign of $\det \Lambda_{d+1}$.

Let us refer to the new matrix that contains the corresponding perturbed parameters as the perturbed matrix, denoted by $\Lambda_{d+1}(\epsilon)$. The modified program that runs on perturbed input will be computing the sign of its determinant, which is given by the following expression.

$$\begin{aligned} &\det \Lambda_{d+1}(\epsilon) \\ &\quad = \det \Lambda_{d+1} + (\epsilon^k \textit{ terms}, 1 \le k \le d-1) + \epsilon^d \begin{vmatrix} 1 & i_1 & i_1^2 & \cdots & i_1^d \\ 1 & i_2 & i_2^2 & \cdots & i_2^d \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & i_{d+1} & i_{d+1}^2 & \cdots & i_{d+1}^d \end{vmatrix}, \end{aligned}$$

where the last term is the determinant of V_{d+1}, a $(d+1) \times (d+1)$ Vandermonde matrix, with

$$\det V_{d+1} = \prod_{\substack{k,l \in \{1,\ldots,d\} \\ k > l}} (i_k - i_l).$$

The second matrix of interest has rows representing input points $p_{i_1}, p_{i_2}, \ldots, p_{i_d}$:

$$\Delta_d = \begin{bmatrix} p_{i_1,1} & p_{i_1,2} & \cdots & p_{i_1,d} \\ p_{i_2,1} & p_{i_2,2} & \cdots & p_{i_2,d} \\ \vdots & \vdots & & \vdots \\ p_{i_d,1} & p_{i_d,2} & \cdots & p_{i_d,d} \end{bmatrix}.$$

This test decides the orientation of points expressed in their homogeneous coordinates. In a dual setting, such as in [11], the input objects are hyperplanes in $(d-1)$-dimensional space and the test indicates on which side of the first hyperplane lies the intersection of the other $d-1$ hyperplanes. Then the determinant vanishes if and only if the d hyperplanes have a nonempty intersection; for this reason, this is called the transversality test.

The corresponding matrix $\Delta_d(\epsilon)$ of the perturbed parameters has determinant

$$\det \Delta_d(\epsilon) = \det \Delta_d + (\epsilon^k \text{ terms}, 1 \le k \le d-1) + \epsilon^d \begin{vmatrix} i_1 & i_1^2 & \cdots & i_1^d \\ i_2 & i_2^2 & \cdots & i_2^d \\ \vdots & \vdots & & \vdots \\ i_d & i_d^2 & \cdots & i_d^d \end{vmatrix},$$

where the coefficient of ϵ^d is the determinant of another Vandermonde matrix U_d and can be expressed as follows:

$$\det U_d = \prod_{k=1}^{d} i_k \quad \det V_d = \prod_{k=1}^{d} i_k \quad \prod_{\substack{k,l=1 \\ k>l}}^{d} (i_k - i_l).$$

LEMMA 4.1. *Given a real RAM program, there exists a positive real constant ϵ_0 such that, for every positive real $\epsilon < \epsilon_0$, every $\Lambda_{d+1}(\epsilon)$ and $\Delta_d(\epsilon)$ matrix occurring at a branching node of the program is nonsingular and its determinant has constant sign.*

Proof. The perturbed determinants are univariate polynomials over the reals. Any polynomial over an ordered field that is not identically zero has an algebraic set of roots of positive codimension. Thus, for univariate polynomials, this set is the union of a finite number of points.

Neither $\det \Lambda_{d+1}(\epsilon)$ nor $\det \Delta_d(\epsilon)$ are identically zero because the highest-order term never vanishes, since all indices are distinct and positive. Hence, there exists a finite number of roots for each symbolic determinant in ϵ. Letting ϵ_0 be the minimum positive such root over all test determinants in the program proves the lemma. □

THEOREM 4.2. *Perturbation* (1) *is valid with respect to algorithms that branch on determinants of $\Lambda_{\delta+1}$ and Δ_δ, for $\delta \leq d$, where d is the dimension of the geometric space in which the input points lie.*

Proof. The perturbed instance is clearly arbitrarily close to the original one since ϵ tends to zero. This instance is also in general position because both kinds of perturbed determinants have a well-defined sign that is never zero for sufficiently small ϵ, from the previous lemma. Finally, since the sign of perturbed determinants is the sign of the lowest-order nonvanishing term, when the original determinant is nonzero it dominates the ϵ-polynomial. Then all branches take the same direction, for a given instance, before and after the perturbation. □

We now address the question of computing the sign of the perturbed determinant. One obvious way is to evaluate the terms in the determinant's ϵ-expansion in increasing order of the exponent of ϵ. The process stops at the first nonvanishing term and reports its sign. This is essentially the approach adopted by SoS and Yap's technique. Our perturbation scheme lends itself to a more efficient trick which reduces the determinant calculation to a characteristic polynomial computation, which also avoids the requirement for symbolic manipulation:

$$\begin{aligned}
\det \Lambda_{d+1}(\epsilon) &= \frac{1}{\epsilon} \begin{vmatrix} \epsilon & p_{i_1,1}(\epsilon) & \cdots & p_{i_1,d}(\epsilon) \\ \vdots & \vdots & & \vdots \\ \epsilon & p_{i_{d+1},1}(\epsilon) & \cdots & p_{i_{d+1},d}(\epsilon) \end{vmatrix} \\
&= \frac{1}{\epsilon} \det \left(\begin{bmatrix} 0 & p_{i_1,1} & \cdots & p_{i_1,d} \\ \vdots & \vdots & & \vdots \\ 0 & p_{i_{d+1},1} & \cdots & p_{i_{d+1},d} \end{bmatrix} + \epsilon\, V_{d+1} \right) \\
&= \frac{1}{\epsilon} \det (L + \epsilon\, V_{d+1}).
\end{aligned}$$

Having implicitly defined L, denoting by I_k the $k \times k$ unit matrix, and relying on the fact that every Vandermonde matrix is invertible, we have

$$\begin{aligned}
\det \Lambda_{d+1}(\epsilon) &= \frac{1}{\epsilon} \det(-V_{d+1}) \det(-(V_{d+1})^{-1} L - \epsilon\, I_{d+1}) \\
&= \frac{1}{\epsilon} (-1)^{d+1} \det V_{d+1} \det(M - \epsilon\, I_{d+1}). \qquad (2)
\end{aligned}$$

Similarly,

$$\det \Delta_d(\epsilon) = \det(\Delta_d + \epsilon U_d) \;=\; \det(-U_d)\, \det(-(U_d)^{-1}\Delta_d - \epsilon I_d)$$
$$= (-1)^d \prod_{k=1}^{d} i_k \;\det V_d \;\det(N - \epsilon\, I_d).$$

Matrices M and N are defined implicitly. Notice that we have reduced the computation of a symbolic determinant to calculating the characteristic polynomial in ϵ of M or N, respectively.

We now prove the efficiency of this approach. Let $MM(k)$ denote the number of multiplications and divisions needed to multiply two $k \times k$ matrices, which is currently $O(k^{2.376})$ [5].

LEMMA 4.3. *Computing the sign of perturbed determinants* $\det \Lambda_{d+1}(\epsilon)$ *and* $\det \Delta_d(\epsilon)$ *can be done in* $O(MM(d) \log d)$ *arithmetic steps.*

Proof. The determinant and the inverse of a $d \times d$ Vandermonde matrix takes at most $O(d^2)$ arithmetic steps [23], while computing M or N as a matrix product takes $O(MM(d))$ operations. Computing $\det(M - \epsilon\, I_{d+1})$ or $\det(N - \epsilon\, I_d)$ is a characteristic polynomial computation for which there exists an algorithm by Keller–Gehrig [16] requiring $O(MM(d) \log d)$ operations. This algorithm is purely numeric as it transforms matrix M or N, respectively, to a new matrix that contains the coefficients of the characteristic polynomial in the last column. □

A brief discussion of modular arithmetic is in order here because, in addition to being the most commonly used method to carry out exact arithmetic on computers, it is also the most economical for computing the perturbed determinants. Let k denote the total number of finite fields required for a particular computation, which is proportional to the bit size of the quantity that is to be eventually computed. Suppose that each finite field $\mathbb{Z}_q$ is defined by a constant-size prime integer q which can be obtained in constant time from an existing and sufficiently long list of primes. Following the exposition in [1], the *first stage* consists of mapping each matrix element into its k residues, the *second stage* performs the particular computation in k different finite fields, and the *third stage* applies the Chinese remainder theorem to find the answer from its k residues. The first and third stages both have bit complexity $O(M(k) \log k)$ while that of the second stage depends on the computation performed. The modular method is applicable to rational inputs with the same asymptotic complexity [7].

Let s be the maximum bit size of any input parameter with $s = \Omega(\log n)$, since we have assumed that a constant fraction of input points is distinct.

THEOREM 4.4. *Consider algorithms that branch on the determinants of* $\Lambda_{\delta+1}$ *and* Δ_δ, *for* $\delta \leq d$, *where* d *is the dimension of the geometric space of the input points. Perturbation* (1) *increases the asymptotic running-time complexity of the algorithm under the algebraic model by* $O(\log d)$. *Under the bit model, the worst-case complexity is increased by a factor of* $O(d^{1+\alpha})$, *where* α *is an arbitrarily small positive constant that accounts for the polylogarithmic factors.*

Proof. The previous lemma proves the claim on the algebraic complexity since the original complexity of computing a $d \times d$ determinant is $\Theta(MM(d))$ [15].

In what follows we concentrate without loss of generality on the sidedness test. In the original setting, the worst-case bit size of the determinant is $\Theta(ds)$ and using modular arithmetic requires $k = \Theta(ds)$ distinct finite fields. The first stage maps d^2 quantities to their respective residues, the second stage computes the determinant modulo some prime q, while the last stage's complexity is dominated. Hence the overall worst-case complexity is

$$\Theta(d^2(ds)^{1+\alpha} + ds\, MM(d)),$$

where α is an arbitrarily small positive constant.

For the perturbed determinant we must compute the coefficients of the characteristic polynomial of M. Observe from (2) that this is a scalar multiple of the ϵ-polynomial $\det \Lambda_{d+1}(\epsilon)$; therefore the latter's coefficient sizes provide upper bounds on the sizes of the characteristic polynomial coefficients. Now, each entry of $\Lambda_{d+1}(\epsilon)$ is a sum of an original point coordinate and a perturbation quantity, hence its bit size is the maximum of s and $d \log n$. All coefficients of $\det \Lambda_{d+1}(\epsilon)$ are sums of determinants of order at most d that have entries of bit size $s + d \log n$. Thus the coefficients have size $O(ds + d^2 \log n)$, which also provides an asymptotic upper bound on the number of finite fields. Hence the new bit complexity is

$$O(d^2(ds + d^2 \log n)^{1+\alpha} + (ds + d^2 \log n) MM(d) \log d),$$

where α is another arbitrarily small positive constant. To complete the proof we apply the asymptotic lower bound on s. □

The section concludes with an application to the beneath-beyond convex hull algorithm for general dimension, presented in [10]. The algorithm is incremental and relies on the hypothesis of general position with respect to the two tests used for branching. The first simply sorts the points along some coordinate assuming that no two points have the same coordinate. The second is essentially the sidedness test, called once the convex hull of a subset of the points is constructed: given a $(d-1)$-dimensional facet and a query point, decide whether the point lies on the same side of the facet as the hull or not. Under perturbation (1) this test can be implemented by at most two sidedness tests.

The perturbation is transparent with respect to the rest of the algorithm. The two branching tests can be thought of as subroutines that are given subsets of input points and return a nonzero sign to avoid degeneracies. The polytope constructed is simplicial, which means that all faces are simplices. Restricting attention to the facets, we note that their number is not minimum because some may be created to subdivide a nonsimplex facet into simplices, while others may include input points in the interior of some facet which have been perturbed into polytope vertices.

Our approach is most favorable in applications where such redundancy is immaterial, for instance, in computing the volume of the convex hull. In this case, under the symmetric volume metric, the problem mapping is continuous and the exact answer is readily obtained by setting $\epsilon = 0$ in the expression of each partial volume that makes up the overall polytope. These partial volumes are $\Lambda_{d+1}(\epsilon)$ determinants computed in the course of the algorithm, so there is no extra cost for calculating the exact volume.

The artificial facets may have to be eliminated for other applications through a postprocessing phase. This involves checking every facet against every one of its d adjacent facets by computing a Λ_{d+1} determinant. If it vanishes, then certain $(d-2)$-dimensional faces must be eliminated and, eventually, certain input points may have to be removed from the vertex set of the convex hull. The algebraic complexity of this phase is asymptotically equal to the product of $dMM(d)$ and the number of facets in the approximate output, hence its bit complexity is dominated by that of the algorithm.

5. Branching on arbitrary rational functions. For the general case where the branching tests are arbitrary rational functions, we propose a randomized perturbation which is easy to implement and applies to algebraic problems such as matrix inversion and linear programming as well as geometric algorithms whose branching functions are not covered by those examined above.

Let f be an arbitrary rational function whose sign determines the direction taken at some branch and express it as p/q, where p, q are polynomials in the input variables, each of total degree bounded by D; recall that D is the maximum total degree in the input variables of any

polynomial in the real RAM program. Suppose that the input variable vector $\mathbf{x}$ belongs to $\mathbb{R}^n$ and let $\mathbf{a} = (a_1, \ldots, a_n)$ be a particular input instance, hence the input size is n.

For a given input, define the perturbed instance $\mathbf{a}(\epsilon) = (a_1(\epsilon), \ldots, a_n(\epsilon))$ as follows:

$$a_i(\epsilon) = a_i + \epsilon\, r_i, \tag{3}$$

where ϵ is an infinitesimal symbolic variable and r_i is a random integer.

Each r_i is chosen uniformly over a range that depends on the desired probability that none of the branching polynomials vanishes. This probability of success can be fixed to be arbitrarily high. It is parameterized by a real constant $c \geq 1$; all claims in this section hold with probability at least $1 - 1/c$.

The total number of polynomials appearing at the numerator or denominator of a branch expression is at most $2 \cdot 3^T$, where T is the maximum number of branches on any execution path. Schwartz's lemma [19] requires that the range of the random values contains at least as many values as the product of c and the total degree of the polynomial whose roots we wish to avoid. Here, this polynomial is the product of all branch polynomials, hence its degree is bounded by $2 \cdot 3^T D$. Therefore, the bit size of the perturbation quantities is

$$\lceil \lg c + \lg D + (\lg 3)\, T + 1 \rceil,$$

where lg denotes the logarithm of base 2.

It is feasible that for some set of random variables, $\mathbf{a}(\epsilon)$ will still cause some branching polynomial to vanish. In this case the perturbation has failed, so the algorithm is restarted and new random variables are picked, independently and uniformly distributed over the same range. It is not clear that any deterministic scheme could avoid the zeros of all polynomials without taking time at least exponential in the number of variables. Intuitively, our method is faster because it randomly selects one n-dimensional perturbation vector instead of trying out all possible ones.

LEMMA 5.1. *Let the entries of* $\mathbf{r} = (r_1, \ldots, r_n)$ *be independently and uniformly chosen integers of* $\lceil \lg c + \lg D + (\lg 3)\, T + 1 \rceil$ *bits each, for any* $c \geq 1$. *Then there exists with probability at least* $1 - 1/c$, *a positive real constant* ϵ_0 *such that, for every positive real* $\epsilon < \epsilon_0$, *every branching rational function* $f(\mathbf{a} + \epsilon\mathbf{r})$ *is defined, nonzero, and of constant sign.*

Proof. Let $g(\mathbf{a} + \epsilon\mathbf{r})$ be any polynomial appearing at the numerator or denominator of some branch expression and let $G(\mathbf{a} + \epsilon\mathbf{r})$ be the product of all distinct polynomials g. By hypothesis, none of these polynomials is identically zero, therefore G also is not identically zero. For a moment, fix $\epsilon = 1$ and consider $G(\mathbf{a} + 1\mathbf{r})$ as a polynomial in $\mathbf{r}$, whose degree in $\mathbf{x}$ and $\mathbf{r}$ is the same. Since D bounds the total degree of any polynomial g, the total degree of G is at most $2 \cdot 3^T D$. Now we apply a lemma proven in [19]. The probability that $\mathbf{r}$, chosen uniformly at random with the given size, is a root of $G(\mathbf{a} + 1\mathbf{r})$ is at most $1/c$. All claims that follow concern the particular $\mathbf{r}$ and hold with probability at least $1 - 1/c$.

First observe that none of the polynomials $g(\mathbf{a} + 1\mathbf{r})$ vanishes at $\mathbf{r}$, hence every $g(\mathbf{a} + \epsilon\mathbf{r})$ may be regarded as a polynomial in ϵ that is not identically zero. Consequently, its zero set is of positive codimension and, more specifically, a finite point set. Consider the minimum positive root for every g and let ϵ_0 be the minimum over all polynomials g. □

THEOREM 5.2. *Perturbation* (3) *is valid with arbitrarily high probability with respect to any algorithm that branches on rational functions in the input variables.*

Proof. The perturbed instance is arbitrarily close to the original one as ϵ tends to zero. Branches decide on the sign of a perturbed rational expression, which is the sign of the lowest-order term in the ϵ-polynomial that does not vanish. By the previous lemma all polynomials have a constant nonzero sign for sufficiently small ϵ, hence $\mathbf{a}(\epsilon)$ is in general position. For non-degenerate inputs all query polynomials have a nonvanishing real part, i.e., a term independent of ϵ which dominates the sign. □

What is the tradeoff in efficiency? The algebraic complexity of the algorithm is increased by the time required to manipulate the ϵ-polynomials symbolically, which depends on D, since the degree of every polynomial in ϵ is the same as its total degree in $\mathbf{x}$. The bit complexity is also affected by D but not by c which is fixed.

LEMMA 5.3. *Under perturbation* (3) *the algebraic time complexity of the branching instructions and the arithmetic operations is* $O(D)$ *and* $O(D \log^2 D)$, *respectively.*

Proof. Each operation in $\{+, -, *, /\}$ involves multiplication of ϵ-polynomials and a GCD computation to reduce to lowest terms so that the degree bound D is observed. The multiplication takes time $O(D \log D)$ and the GCD $O(D \log^2 D)$ [1]. Branching instructions must find the lowest nonvanishing term in the corresponding ϵ-polynomial, which takes $O(D)$ time. □

Degree D cannot be bounded in general by a polynomial in the algebraic complexity, which implies that the perturbation may be prohibitively expensive under the algebraic model. Exponentiating a rational number, for instance, takes roughly a logarithmic number of steps in the exponent, while on perturbed input the worst-case algebraic complexity is at least linear in it, which means the complexity overhead is exponential in the original complexity. However, we obtain better bounds by considering bit complexities.

THEOREM 5.4. *Under the algebraic model, the running time increases due to perturbation* (3) *by a multiplicative factor of* $O(D^{1+\alpha})$, *where* D *is the maximum total degree in the input variables of any polynomial in the real RAM program and* α *is an arbitrarily small positive constant accounting for the polylogarithmic factor. Under the bit model, the overhead for the worst-case complexity is* $O(\phi^{2+\alpha}(n, s))$, *where* $\phi(n, s)$ *asymptotically bounds the original worst-case bit complexity of the algorithm,* s *is the maximum bit size of the input quantities, and* α *is an arbitrarily small positive constant.*

Proof. By the previous lemma for every instruction the overhead is $O(D \log^2 D)$. This establishes the algebraic complexity overhead.

By the definition of T there exists an execution path with bit complexity $\Omega(T)$, hence $\phi(n, s) = \Omega(T)$. By the definition of D, there exists a path where a polynomial of degree D in the input variables is computed. Since the only legal arithmetic operations lie in $\{+, -, *, /\}$, there must exist an earlier operation on this path computing a polynomial of degree at least $D/2$, hence computing values of bit size $sD/2$. The operation that uses these values as operands has bit cost $\Omega(sD/2) = \Omega(D)$. Hence $\phi(n, s) = \Omega(D)$.

After the perturbation, the same program operates on perturbed quantities; their starting bit size is multiplied by $O(\log D + T)$, assuming c is constant. Moreover, the algebraic complexity has overhead $O(D \log^2 D)$. Hence, the worst-case bit complexity overhead is

$$O((\log D + T) D \log^2 D). \tag{4}$$

Recalling the two lower bounds on $\phi(n, s)$, we have $\phi(n, s)^{2+\alpha} = \Omega(T D^{1+\alpha})$, which bounds the bit complexity overhead (4), for some appropriate $\alpha > 0$. □

COROLLARY 5.5. *Perturbation* (3) *does not affect the worst-case bit complexity class of the algorithm. In particular, if the original complexity lies in P or EXPTIME, then the complexity on perturbed input also lies in P or EXPTIME, respectively.*

Proof. The proof is immediate from the previous theorem. □

Taking up the running example of Gaussian elimination for the matrix inversion problem, we observe that no checks for zero denominators have to be carried out on perturbed input. Perturbation thus eliminates the need for interchanging rows. Computation is symbolic, with GCD operations at every step to cancel common terms and thus prohibit the degree in ϵ of the symbolic polynomials from growing exponentially in the number of examined rows. For nonsingular matrices, the result is obtained by setting ϵ to zero at the end. For singular

instances, this causes some denominator to vanish, so we take the limit as ϵ goes to zero. For singular matrices the result is some real matrix that approximates, in a sense, the inverse.

6. Conclusion. We studied algorithms modeled as programs on real RAMs with inputs from an infinite ordered field and described perturbations on the input, such that an algorithm designed under the assumption of nondegeneracy can be applied to all inputs. Our perturbations satisfy the validity condition set out in §2 which guarantees the relevance of the output with respect to the initial problem.

We defined a deterministic method for algorithms with determinant tests and a randomized one for arbitrary test functions. The first applies to algorithms from computational geometry whose branching tests can be expressed as a determinant of a Λ or Δ matrix. Ignoring polylogarithmic factors in the geometric dimension, the deterministic scheme does not affect the algebraic complexity but incurs an overhead to the worst-case bit complexity that is linear in the dimension. The second perturbation, applicable to most geometric and algebraic algorithms, incurs a worst-case overhead under the bit model that is bounded by a small-degree polynomial in the original complexity. Both methods are characterized by their conceptual simplicity and are significantly faster than previous ones.

Examining branching tests that come up in other geometric algorithms and trying to improve on efficiency are natural extensions to this work, partly fulfilled in [14]. It is also interesting to attempt to extend the notion of degeneracy over finite fields, where the lack of order makes our definition of degeneracy invalid. Another direction of generalization is to observe that each leaf subspace is associated with a semialgebraic set defined by the branch polynomials on the respective execution paths. We may wish to perturb these sets into general position.

Acknowledgment. We wish to thank K. Mehlhorn for several useful comments.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] L. Blum, M. Shub, and S. Smale, *On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines*, Bull. Amer. Math. Soc., 21 (1989), pp. 1–46.

[3] J. Bochnak, M. Coste, and M. F. Roy, *Géométrie Algébrique Réelle*, Ergebnisse der Mathematik 3, No. 12, Springer-Verlag, Berlin, 1987.

[4] J. F. Canny, *Computing roadmaps of semi-algebraic sets*, Proc. 9th Symposium on Applied Algebra, Algebraic Algorithms and Error-Corr. Codes, Lecture Notes in Computer Science 539, Springer-Verlag, Berlin, 1991, pp. 94–107.

[5] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, J. Symbolic Comput., 9 (1990), pp. 251–280.

[6] G. B. Dantzig, *Linear Programming and Extensions*, Princeton University Press, Princeton, 1963.

[7] J. H. Davenport, Y. Siret, and E. Tournier, *Computer Algebra*, Academic Press, London, 1988.

[8] K. Dobrindt, K. Mehlhorn, and M. Yvinec, *A complete framework for the intersection of a general polyhedron with a convex one*, Proc. 3rd Workshop Algorithms Data Struct., Lecture Notes in Computer Science 709, Springer-Verlag, Berlin, 1993, pp. 314–324.

[9] H. Edelsbrunner, *Edge-skeletons in arrangements with applications*, Algorithmica, 1 (1986), pp. 93–109.

[10] ———, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Heidelberg, 1987.

[11] H. Edelsbrunner and L. J. Guibas, *Topologically sweeping an arrangement*, Proc. 18th ACM Symposium on Theory of Computing, 1986, pp. 389–403.

[12] H. Edelsbrunner and E. P. Mücke, *Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms*, ACM Trans. Graphics, 9 (1990), pp. 67–104.

[13] H. Edelsbrunner and R. Waupotitsch, *Computing a ham-sandwich cut in two dimensions*, J. Symbolic Comput., 2 (1986), pp. 171–178.

[14] I. Emiris and J. Canny, *An efficient approach to removing geometric degeneracies*, Proc. 8th ACM Symposium on Computational Geometry, 1992, pp. 74–82.

[15] J. VON ZUR GATHEN, *Algebraic complexity theory*, in Annual Review of Computer Science, Vol. 3, J. Traub, ed., Annual Reviews, Palo Alto, 1988, pp. 317–347.

[16] W. KELLER-GEHRIG, *Fast algorithms for the characteristic polynomial*, Theoret. Comput. Sci., 36 (1985), pp. 309–317.

[17] C. MONMA, M. PATERSON, S. SURI, AND F. YAO, *Computing euclidean maximum spanning trees*, Proc. 4th ACM Symposium on Computational Geometry, 1988, pp. 241–251.

[18] F. P. PREPARATA AND M. I. SHAMOS, *Computational Geometry*, Springer-Verlag, New York, 1985.

[19] J. T. SCHWARTZ, *Fast probabilistic algorithms for verification of polynomial identities*, J. Assoc. Comput. Mach., 27 (1980), pp. 701–717.

[20] A. TARSKI, *A Decision Method for Elementary Algebra and Geometry*, University of California Press, Berkeley, 1948.

[21] C.-K. YAP, *Symbolic treatment of geometric degeneracies*, J. Symbolic Comput., 10 (1990), pp. 349–370.

[22] ———, *A geometric consistency theorem for a symbolic perturbation scheme*, J. Comput. System Sci., 40 (1990), pp. 2–18.

[23] R. ZIPPEL, *Interpolating polynomials from their values*, J. Symbolic Comput., 9 (1990), pp. 375–403.

SIAM J. COMPUT.
Vol. 24, No. 3, pp. 665–671, June 1995

015

A PRIORI BOUNDS ON THE EUCLIDEAN TRAVELING SALESMAN*

TIMOTHY LAW SNYDER† AND J. MICHAEL STEELE‡

Abstract. It is proved that there are constants c_1, c_2, and c_3 such that for any set S of n points in the unit square and for any minimum-length tour T of S (1) the sum of squares of the edge lengths of T is bounded by $c_1 \log n$, (2) the number of edges having length t or greater in T is at most c_2/t^2, and (3) the sum of edge lengths of any subset E of T is bounded by $c_3|E|^{1/2}$. The second and third bounds are independent of the number of points in S, as well as their locations. Extensions to dimensions $d > 2$ are also sketched. The presence of the logarithmic term in (1) is engaging because such a term is not needed in the case of the minimum spanning tree and several analogous problems, and, furthermore, we know that there always exists *some* tour of S (which perhaps does not have minimal length) for which the sum of squared edges is bounded independently of n.

Key words. Euclidean traveling salesman problem, inequalities, squared edge lengths, long edges

AMS subject classifications. 68R10, 05C45, 90C35, 68U05

1. Introduction.

The purpose of this note is to provide a priori bounds on quantities related to the edge lengths of an optimal traveling salesman (minimum-length) tour through n points in the unit square. By *a priori* we mean that the bounds are independent of the locations of the points.

Studies of a priori bounds were initiated by Verblunsky (1951) and Few (1955). Few showed that for any set S of n points in the unit square, the length of an optimal traveling salesman tour of S is at most $\sqrt{2n} + 1.75$. Few's result led to a series of improvements, culminating in Karloff (1989), where it was shown that Few's constant could be reduced to less than $\sqrt{2}$. Goddyn (1990) improved similar results in higher dimensions. Our results continue in this tradition by giving a priori inequalities for three other quantities related to the edge lengths of an optimal traveling salesman tour.

The interest in and subtlety of our inequalities comes from the fact that, in contrast to the minimum spanning tree (MST) problem, optimal solutions to the traveling salesman problem (TSP) are not invariant under monotone transformations of the edge weights. Before giving further details on this connection and other related work, we state our main results. We let $|e| = |x - y|$ denote the Euclidean length of the edge $e = xy$ with vertices x and y in $\mathbb{R}^2$ and, in settings where the order of the edges of an optimal tour is not important, we represent a traveling salesman tour by the edge set $\{e_1, e_2, \ldots, e_n\}$. In what follows, an "optimal" traveling salesman tour is a tour that is of minimum length when using Euclidean edge weights.

Our first theorem bounds the sum of squared edge lengths of any optimal traveling salesman tour.

THEOREM 1. *There exists a constant $0 < c_1 < \infty$ such that if $T = \{e_1, e_2, \ldots, e_n\}$ is an optimal traveling salesman tour of $\{x_1, x_2, \ldots, x_n\} \subset [0, 1]^2$ and if $n \geq 2$, then*

$$\sum_{i=1}^{n} |e_i|^2 \leq c_1 \log n. \tag{1.1}$$

Theorem 2 is a bound on the number of edges that are of length t or greater.

*Received by the editors February 24, 1992; accepted for publication (in revised form) January 21, 1994.

†Department of Computer Science, Georgetown University, Washington, DC 20057. The research of this author was supported in part by Georgetown University 1991 Summer Research Award, Georgetown University 1992 Junior Faculty Research Fellowship, and Georgetown College John R. Kennedy, Jr. Faculty Research Fund.

‡Department of Statistics, The Wharton School, University of Pennsylvania, Philadelphia, Pennsylvania 19104. The research of this author was supported in part by the following grants: National Science Foundation grant DMS88-12868 and National Security Agency grants MDA904-89-H-2034, AFOSR-89-0301, and DAAL03-89-G-0092.

THEOREM 2. *There exists a constant* $0 < c_2 < \infty$ *such that, if* $v(n, t)$ *is the number of* $e_i \in T$ *such that* $|e_i| \geq t$, *then for all* $t > 0$ *and* $n \geq 1$,

$$v(n, t) \leq c_2/t^2. \tag{1.2}$$

Theorem 3 gives a bound on the total length of any k-edge subset of an optimal TSP tour.

THEOREM 3. *There exists a constant* $0 < c_3 < \infty$ *such that for each* $E = \{e_{i_1}, e_{i_2}, \ldots, e_{i_k}\} \subseteq T$, *we have*

$$\sum_{i \in E} |e_i| \leq c_3 \sqrt{k}. \tag{1.3}$$

It is interesting to compare these results to their minimum spanning tree analogues. Steele and Snyder (1989) proved MST analogues to (1.2) and (1.3), but these proofs were predicated on a solution to the MST problem via a greedy algorithm and thus were not applicable to the TSP. The best TSP analogue to (1.2) was therefore $v_{\text{TSP}}(n, t) \leq c_{\text{TSP}} \sqrt{n}/t$, for some constant c_{TSP}. The bounds (1.2) and (1.3), however, are independent of n, the number of points, as well as the locations of the points. For this reason, we say that (1.2) and (1.3) are *fully a priori* inequalities.

Inequalities like (1.1) are important in simulations and investigations in which square root computations required for Euclidean lengths are deemed to be too expensive (cf. the discussion in Steele (1990)). It was observed in Steele (1990) in an application of the space-filling curve heuristic that one could obtain a result like (1.1) for the MST, but without the logarithmic factor. Although this result might make the logarithmic term of (1.1) seem disappointing, it is actually best possible since Bern and Eppstein (1993) recently showed that there exist a positive constant c and point sets S with $|S| \to \infty$ such that $\sum_{e \in T} |e|^2 \geq c \log |S|$.

Part of the interest in these results comes from the fact that there are closely connected inequalities that exhibit strikingly different behavior from the optimal TSP tour. In particular, there is a constant c' and for all n there is a nonminimal length tour T' of S with $|S| = n$ such that, for all $n \geq 2$, $\sum_{e \in T'} |e|^2 \leq c'$. These tours can be obtained via the space-filling heuristic as noted in the discussion of the MST. Neumann (1982) showed that such tours can be obtained by appropriately generalizing the Pythagorean theorem, a construction that, upon reflection, almost parallels that of some space-filling curves.

The curious issue for the TSP is that although there is *some* tour T' that makes $\sum_{e \in T'} |e|^2$ particularly small, the Bern and Eppstein (1993) result tells us that a traveling salesman tour T minimizing $\sum_{e \in T} |e|$ need not do nearly so well. Because of the matroidal properties of the MST, these issues do not arise in its analysis; analyzing the optimal TSP is more difficult.

In the final section, we will comment further on this as well as problems concerning points in $[0, 1]^d$ for dimension $d > 2$. In §2, we prove a technical result that is applied in §3 to prove our main results.

2. Edge lemmas. The second lemma of this section explicates a property of edges in a TSP tour that will be useful in the next section, where we prove our main results. Our first lemma gives a simple geometric bound concerning diagonals of quadrilaterals. In the statement of Lemma 1, the term "diagonal" is used to denote a segment connecting nonadjacent vertices of a quadrilateral, regardless of whether the quadrilateral is convex.

LEMMA 1. *Let* L_1 *and* L_2 *be two nonintersecting line segments satisfying* $r \leq |L_i| \leq \beta r$, *where* $\beta > 1$ *and* $r > 0$. *Suppose the midpoints of* L_1 *and* L_2 *are separated by distance* λ, *where* $\lambda \leq \min\{\frac{1}{2}|L_1|, \frac{1}{2}|L_2|\}$. *If the endpoints of* L_1 *and* L_2 *are joined to form a quadrilateral with sides* L_1, L_2, S_1, *and* S_2, *then* $|S_i| \leq \frac{1}{2}(\beta - 1)r + 3\lambda$ *for* $i = 1, 2$.

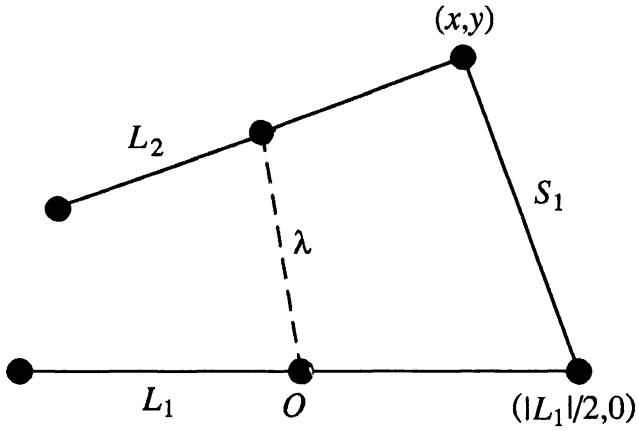


FIG. 1. *The lines L_1 and L_2 in the proof of Lemma* 1.

Proof. Without loss of generality, we can assume that L_1 is the longer of the two lines and that it is oriented along the x axis with its midpoint at the origin. We can also assume that L_2 lies entirely in the upper half plane. Let (x, y) denote the rightmost endpoint of L_2 and let S_1 be the line segment determined by (x, y) and $(|L_1|/2, 0)$ (see Fig. 1, which illustrates a convex quadrilateral). By the triangle inequality, the segment from the origin to (x, y) is at most $\frac{1}{2}|L_2| + \lambda$, so $x \leq \frac{1}{2}|L_2| + \lambda$.

We also claim that $\frac{1}{2}|L_2| - \lambda \leq x$. To prove the claim, consider the disk D of radius λ centered at the midpoint m of L_2. Since $\lambda \leq \frac{1}{2}|L_2|$, the point (x, y) must lie outside the interior of D. Since L_2 lies entirely in the upper half plane, the endpoints of L_2 must lie in the shaded regions in Fig. 2, with (x, y) constrained to lie in the first quadrant. Letting $(x,' 0)$ be the point where the x axis intersects the circle with center m passing through (x, y), it is clear from the figure that $x \geq x'$. However, the origin–x' segment is greater than or equal to $\frac{1}{2}|L_2| - \lambda$ since $\frac{1}{2}|L_2| - \lambda$ is the minimum distance from $(x,' 0)$ to D. This proves the claim and yields

$$\frac{1}{2}|L_2| - \lambda \leq x \leq \frac{1}{2}|L_2| + \lambda. \tag{2.1}$$

Since L_1 and L_2 do not intersect, $0 < y < 2\lambda$. Combining this with (2.1) gives us $|S_1| \leq |x - \frac{1}{2}|L_1|| + y \leq \frac{1}{2}(\beta - 1)r + 3\lambda$, as claimed. □

LEMMA 2. *Let $\{e_1, e_2, \ldots, e_n\}$ denote the edges of an optimal traveling salesman tour of $\{x_1, x_2, \ldots, x_n\} \subset \mathbb{R}^2$. For each e_i satisfying $r \leq |e_i| \leq \beta r$, where $r > 0$, let D_i denote the disk of radius $\alpha|e_i|$ centered at the midpoint of e_i, where $\alpha = 1/13$ and $\beta = 3/2$. Then, for any three disks D_{i_1}, D_{i_2}, and D_{i_3}, the intersection $D_{i_1} \cap D_{i_2} \cap D_{i_3}$ is empty.*

Proof. Suppose at first that $D_{i_1} \cap D_{i_2} \cap D_{i_3} \neq \emptyset$ and that the edges e_{i_1}, e_{i_2}, and e_{i_3} share no common vertex. Without loss of generality, let $i_j = j$ for $j = 1, 2, 3$. We show that if D_1, D_2, and D_3 have a point in common, then it is possible to construct a shorter tour through $\{x_1, x_2, \ldots, x_n\}$. It is well known that edges of an optimal Euclidean traveling salesman tour cannot intersect. We can therefore assume that $e_1 = a_1b_1$, with midpoint m_1 and endpoint a_1 to the left and b_1 to the right, is oriented along the x axis. Similarly we can assume the midpoint m_2 of $e_2 = a_2b_2$ lies above e_1 and the midpoint m_3 of $e_3 = a_3b_3$ lies above e_2, as illustrated in Fig. 3.

Since the endpoints of the e_i, $\{a_1, a_2, a_3, b_1, b_2, b_3\}$, are distinct and are on the tour, there is a pair a_i, b_j with $i \neq j$ such that a_i and b_j are joined by a path that contains none of the

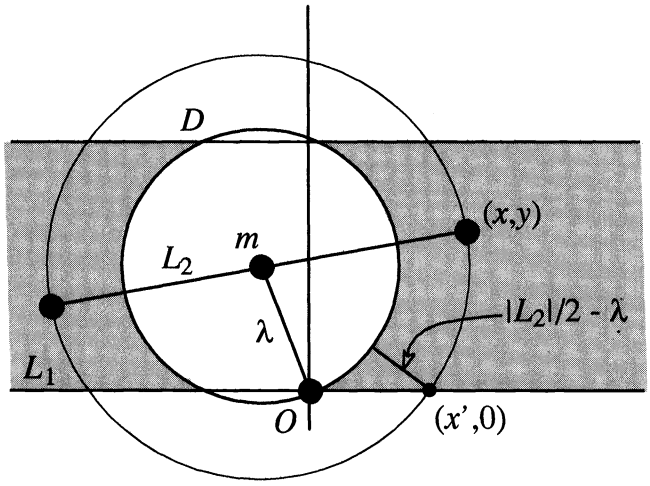


FIG. 2. *The disk D and its relation to x, L_2, and λ.*

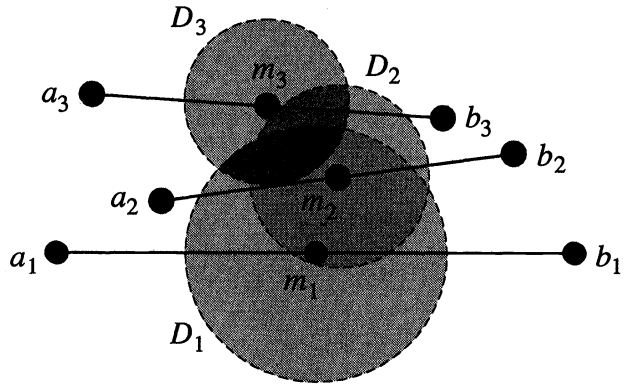


FIG. 3. *Three nonintersecting lines of a TSP tour and their D_i. Here, $\alpha = 1/2$ for visual clarity.*

edges e_1, e_2, and e_3. We now claim that we can construct a shorter tour by replacing edges e_i and e_j with edges a_ia_j and b_ib_j. This contradiction will establish the lemma.

For specificity, assume that $i = 2$ and $j = 3$, as shown in Fig. 4. We form a new path from a_2 to b_3 by deleting e_2 and e_3 and adding the edges a_2a_3 and b_2b_3. Since D_1, D_2, and D_3 have a point in common, the midpoints of e_2 and e_3 can be separated by at most the summed radii of D_2 and D_3, which is $\alpha|e_2| + \alpha|e_3|$. Setting $\lambda = \alpha(|e_2| + |e_3|)$ and recalling that $r \le |e_i| \le \beta r$, we note that $\lambda \le \alpha(|e_2| + \beta|e_2|) \le \frac{1}{2}|e_2|$; similarly, $\lambda \le \frac{1}{2}|e_3|$. In addition, we have $\lambda \le 2\alpha\beta r$. These facts allow us to apply Lemma 1 to estimate the net change Δ in the path length as

$$\begin{aligned}
\Delta &= |a_2 - a_3| + |b_2 - b_3| - |e_2| - |e_3| \\
&\le 2\left[\frac{1}{2}(\beta - 1)r + 3\lambda\right] - 2r \\
&\le 2\left[\frac{1}{2}(\beta - 1)r + 6\alpha\beta r\right] - 2r \\
&= (\beta - 3 + 12\alpha\beta)r.
\end{aligned}$$

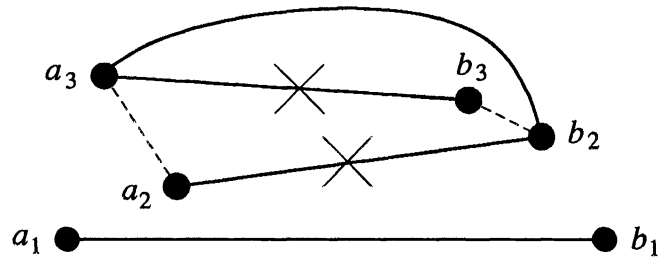


FIG. 4. *Rebuilding the a_2 to b_3 path when $i = 2$ and $j = 3$ in Lemma 2. The curved arc is a path, the $\times$'ed edges have been removed, and the dashed edges have been added.*

The choices $\beta = 3/2$ and $\alpha = 1/13$ guarantee that $\Delta < 0$.

For the case of $i = 1$ and $j = 3$, one obtains identical bounds on the change in the tour length when replacing e_i and e_j with $a_i a_j$ and $b_i b_j$. Without loss of generality, the $i = 2$, $j = 3$ and $i = 1$, $j = 3$ cases are the only cases that need to be considered.

To complete the proof, note that any vertex shared by any of e_{i_1}, e_{i_2}, and e_{i_3} can be replaced with two vertices that are viewed as being joined by an edge of length 0. The above analysis can then be applied as before without change to obtain a contradiction. □

3. A priori edge-length bounds. We are now in position to prove our main results. Label the edges of an optimal tour T of $\{x_1, x_2, \ldots, x_n\} \subset [0, 1]^2$ in order as $e_1, e_2, \ldots, e_n$. We first construct disks D_i of radius $\alpha|e_i|$ and center at the midpoint of e_i for each $1 \le i \le n$, where $\alpha = 1/13$. Let $\psi_i(\cdot)$ denote the indicator function of D_i, i.e., for all $x \in \mathbb{R}^2$, $\psi_i(x) = 1$ if $x \in D_i$; otherwise $\psi_i(x) = 0$. Let A be the set of all i such that $r \le |e_i| \le \beta r$, where $\beta = 3/2$. We then claim that

$$\sum_{i \in A} \psi_i(x) \le 2\psi(x), \tag{3.1}$$

where $\psi(\cdot)$ is the indicator function of the square $[-1, 2]^2$.

To prove the claim, note that for $\beta = 3/2$ and $\alpha = 1/13$, Lemma 2 tells us that no three disks of A intersect. Hence, the point $x \in \mathbb{R}^2$ can belong to at most two disks associated with A. Furthermore, since any disk with center in $[0, 1]^2$ and radius bounded by $\alpha\beta r$ is contained in $[-\alpha\beta r, 1 + \alpha\beta r]^2 \subset [-1, 2]^2$, we need only concern ourselves with the square $x \in [-1, 2]^2$. This proves the claim.

If we now integrate (3.1) over x, we obtain a basic bound on a subset of the squared edge lengths of an optimal TSP tour:

$$\sum_{r \le |e_i| \le \beta r} |e_i|^2 \le c, \tag{3.2}$$

where $c = 18\alpha^{-2}\pi^{-1}$. The bound is then used as follows.

$$\begin{aligned} \sum_{i=1}^{n} |e_i|^2 &\le 1 + \sum_{n^{-1/2} \le |e_i| \le \sqrt{2}} |e_i|^2 \\ &\le 1 + \sum_{k=1}^{m} \sum_{\beta^{k-1} n^{-1/2} \le |e_i| \le \beta^k n^{-1/2}} |e_i|^2, \end{aligned} \tag{3.3}$$

where m is the least integer k such that $\beta^k n^{-1/2} \geq \sqrt{2}$. It suffices to take $m = \lceil \log_{3/2}(\sqrt{2n}) \rceil$; applying (3.2) to (3.3) yields the bound

$$\sum_{i=1}^{n} |e_i|^2 \leq c_1 \log n, \tag{3.4}$$

where c_1 is constant as required by Theorem 1. We remark that explicit constants have been given only to facilitate checking; there is little hope of obtaining the best possible bounds on c_1 and related values.

Returning to (3.1) and integrating, we see that since $|e_i| \geq r$ for all $i \in A$,

$$|A| \pi \alpha^2 r^2 \leq 18. \tag{3.5}$$

However, $|A| = \bigl|\{ i : r \leq |e_i| \leq \beta r \}\bigr|$, so for $c = 18/(\pi\alpha^2)$, we have

$$\bigl|\{ i : r \leq |e_i| \leq \beta r \}\bigr| \leq c r^{-2}. \tag{3.6}$$

We can now bound

$$\begin{aligned} \nu(n,t) &= \bigl|\{ i : |e_i| \geq t \}\bigr| \\ &\leq \sum_{k=0}^{m_t - 1} \bigl|\{ i : \beta^k t \leq |e_i| \leq \beta^{k+1} t \}\bigr|, \end{aligned} \tag{3.7}$$

where $m_t = \min_j \{\beta^j t \geq \sqrt{2}\}$. We then use (3.6) to obtain

$$\begin{aligned} \nu(n,t) &\leq c \sum_{k=0}^{m_t - 1} (\beta^k t)^{-2} \\ &\leq c\, t^{-2} \sum_{k=0}^{\infty} \beta^{-2k} \\ &= \frac{c}{1 - \beta^{-2}}\, t^{-2}, \end{aligned} \tag{3.8}$$

which is Theorem 2, with $c_2 = c\beta^2/(\beta^2 - 1)$.

Theorem 3 now results from (3.8) by first noting that $n - \nu(n,x)$ is the number of edges in T of length less than x, then writing

$$\begin{aligned} \sum_{e_i \in E} |e_i| &= \sum_{\substack{e_i \in E \\ |e_i| < t}} |e_i| + \sum_{\substack{e_i \in E \\ |e_i| \geq t}} |e_i| \\ &\leq \sum_{\substack{e_i \in E \\ |e_i| < t}} |e_i| + \sum_{\substack{e_i \in T \\ |e_i| \geq t}} |e_i| \\ &\leq t|E| + \int_t^{\sqrt{2}} x\, d(n - \nu(n,x)) \\ &\leq t|E| - \int_t^{\sqrt{2}} x\, d\nu(n,x). \end{aligned} \tag{3.9}$$

Integrating the rightmost term of (3.9) by parts and then applying (3.8), we obtain

$$
\begin{aligned}
-\int_t^{\sqrt{2}} x\, dv(n,x) &= tv(n,t) + \int_t^{\sqrt{2}} v(n,x)\, dx \\
&\le \frac{c_2}{t} + \int_t^{\infty} \frac{c_2}{x^2}\, dx \\
&\le \frac{2c_2}{t}.
\end{aligned} \tag{3.10}
$$

Inserting (3.10) into (3.9) and setting $t = |E|^{-1/2}$ yields Theorem 3, with $c_3 = 1 + 2c_2$.

4. Concluding remarks. The preceding arguments can be generalized without difficulty to higher dimensions. The key idea is that in Lemma 2 we showed that if three of the D_i associated with edges of a TSP tour had a point in common, then we could find three edges e_1, e_2, and e_3 that were close together and nearly parallel.

We can obtain a proper analogue in dimensions $d > 2$ if we consider the possibility that a large number $N(d)$ of d-spheres $D_i = D(m_i, \alpha|e_i|) \subset \mathbb{R}^d$ intersect and exploit the fact that the surface of any sphere in $\mathbb{R}^d$ can be covered with a finite number $M(\epsilon)$ of spherical caps with polar angle ϵ. In summary, one can prove the following theorem.

THEOREM. *There exist positive constants c_d and c'_d such that for any traveling salesman tour T of $\{x_1, x_2, \ldots, x_n\} \subset [0,1]^d$ and for all $n \ge 2$,*

$$\sum_{e \in T} |e|^d \le c_d \log n, \tag{4.1}$$

and

$$v_d(t) = \big|\{ e \in T : |e| \ge t \}\big| \le c'_d / t^d. \tag{4.2}$$

Furthermore, there exists a positive constant c''_d such that for any $E = \{e_{i_1}, e_{i_2}, \ldots, e_{i_k}\} \subseteq T$, we have

$$\sum_{i \in E} |e_i| \le c''_d k^{(d-1)/d}. \tag{4.3}$$

Acknowledgment. We thank a kind and meticulous referee who corrected some errors and who provided useful suggestions for simplifying our proof of Lemma 2.

REFERENCES

[1] M. BERN AND D. EPPSTEIN (1993), *Worst-case bounds for subadditive geometric graphs*, Proceedings of the Ninth Annual Symposium on Computational Geometry, Association for Computing Machinery, pp. 183–188.

[2] L. FEW (1955), *The shortest path and the longest road through n points in a region*, Mathematika, 2, pp. 141–144.

[3] L. GODDYN (1990), *Quantizers and the worst-case Euclidean traveling salesman problem*, J. Combin. Theory, Ser. B, 50, pp. 65–81.

[4] H. J. KARLOFF (1989), *How long can a Euclidean traveling salesman tour be?*, SIAM J. Discrete Math., 2, pp. 91–99.

[5] D. J. NEUMANN (1982), *A Problem Seminar*, Springer-Verlag, New York, NY.

[6] J. M. STEELE (1990), *Probabilistic and worst-case analyses of classical problems of combinatorial optimization in Euclidean space*, Math. Oper. Res., 15, pp. 749–770.

[7] J. M. STEELE AND T. L. SNYDER (1989), *Worst-case growth rates of some problems from combinatorial optimization*, SIAM J. Comput., 18, pp. 278–287.

[8] S. VERBLUNSKY (1951), *On the shortest path through a number of points*, Proc. Amer. Math. Soc., 2, pp. 904–913.

SIAM J. COMPUT.
Vol. 24, No. 4, pp. 673–681, August 1995

SPARSE REDUCES CONJUNCTIVELY TO TALLY*

HARRY BUHRMAN†, EDITH HEMASPAANDRA‡, AND LUC LONGPRÉ§

Abstract. Polynomials over finite fields are used to show that any sparse set can conjunctively reduce to a tally set. This leads to new results and to simple proofs of known results about various classes that lie between P and P/poly.

Key words. low density sets, conjunctive reductions, truth table reductions, Kolmogorov complexity

AMS subject classifications. 68Q05, 68Q30

1. Introduction.

Sparse sets and tally sets have been the subject of much recent research in structural complexity theory. A thorough survey of results on this topic can be found in [HOW92].

Sparse sets are closely linked to nonuniform complexity classes and circuit complexity. It is well known that sets Turing reducible to sparse sets are those sets that have polynomial size circuits, which is also the same as the advice class P/poly, the class of sets solvable with polynomial size advice. Since sparse sets can be encoded easily as tally sets, this is also the same as the class of sets Turing reducible to tally sets.

For a reduction $\leq_r^p$ and a class of sets $\mathcal{C}$, let $R_r(\mathcal{C})$ be the class of all sets that are $\leq_r^p$-reducible to a set in $\mathcal{C}$. In this terminology, P/poly $= R_T(\text{SPARSE}) = R_T(\text{TALLY})$. There is an interesting structure of sets lying between P and P/poly that can be defined by changing the Turing reductions to weaker reductions, and/or by considering tally sets instead of sparse sets.

The study of the $R_r(\text{SPARSE})$ and $R_r(\text{TALLY})$ classes, for various reductions r, was initiated by Book and Ko in [BK88]. A more extensive study of these classes can be found in [Ko89], [AHOW92], and [AHH+93]. Our main result refutes one of Ko's conjectures [Ko89] by showing that every sparse set is conjunctive truth-table reducible to a tally set as follows:

$$\text{SPARSE} \subseteq R_{ctt}(\text{TALLY}).$$
$$R_{ctt}(\text{SPARSE}) = R_{ctt}(\text{TALLY}).$$

The reduction uses polynomials over finite fields to encode any sparse set into a tally set in such a way that a polynomial-time algorithm can compute membership in the sparse set using a conjunctive truth-table query. This encoding method itself found more applications. Recently, it has been used to show an upward separation for FewP [RRW94]. The more classic encoding method did not seem to work there. It has also been used to handle bottlenecks in neural networks [Wat].

Our result is surprising since it is false for disjunctive truth-table reductions [Ko89]—$\text{SPARSE} \not\subseteq R_{dtt}(\text{TALLY})$—and since it was believed to be false by those who looked at the problem. One way to interpret the result is as follows. It is easy to see that one can encode a sparse set into a tally set. But can it be encoded in such a way that all the information about

*Received by the editors April 2, 1993; accepted for publication (in revised form) January 24, 1994.

†Centrum voor Wishkunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands. This research was done in part while this author was at the University of Amsterdam and visiting Boston University. Supported in part by NSF grant CCR-8814339 and NWO grant SIR 13-603 (buhrman@cwi.nl).

‡Department of Computer Science, Le Moyne College, Syracuse, New York 13214. This research was done in part while this author was at the University of Amsterdam (edith@bamboo.lemoyne.edu).

§Computer Science Department, University of Texas at El Paso, El Paso, Texas 79968 (longpre@cs.utep.edu). This research was done while this author was at Northeastern University and supported in part by National Science Foundation grant CCR-9211174.

the sparse set can be retrieved with a conjunctive truth-table? With a disjunctive truth-table? The answers are yes and no, respectively.

The results allow us to derive corollaries that either settle other open problems or provide simple proofs of previously known results. For example, as a derived result, we refuse another conjecture of Ko by showing

$$R_{bdtt}(\text{SPARSE}) \subseteq R_{ctt}(\text{SPARSE}).$$

These two results and the following result by Gavaldà and Watanabe settle all the remaining open problems from [Ko89]. We end this section by looking at *positive* truth-table reductions [Sel82] ($\leq^p_{ptt}$) to sparse and tally sets. In particular, we show that ptt reductions to tally sets capture the class $R_{tt}(\text{TALLY})$:

$$R_{ptt}(\text{TALLY}) = R_{tt}(\text{TALLY}),$$

and thus

$$R_{ptt}(\text{SPARSE}) = R_{tt}(\text{SPARSE}).$$

In [GW93], Gavaldà and Watanabe use a technique based on Kolmogorov complexity to prove the conjecture of Ko that $R_{ctt}(\text{SPARSE}) \not\subseteq R_{dtt}(\text{SPARSE})$. Their construction actually provides something stronger. If $f(n)$ is an unbounded function from integers to integers, such that $f(n)$ is computable in time polynomial in n, then their construction provides a set that is not $\leq^p_{dtt}$-reducible to any sparse set but is $\leq^p_{ctt}$-reducible to a sparse set using only $f(n)$ queries on inputs of length n: $R_{f(n)\text{-}ctt}(\text{SPARSE}) \not\subseteq R_{dtt}(\text{SPARSE})$, for any polynomial-time-computable unbounded function f. By improving their technique, we are able to make the set reducible to a *tally* set. For any polynomial-time-computable unbounded function f,

$$R_{f(n)\text{-}ctt}(\text{TALLY}) \not\subseteq R_{dtt}(\text{SPARSE}).$$

Combining this with our main result allows us to strengthen one of Ko's results and show that for any polynomial-time-computable unbounded function f,

$$R_{f(n)\text{-}dtt}(\text{TALLY}) \not\subseteq R_{ctt}(\text{SPARSE}).$$

This is optimal in some sense and reveals the following picture: $R_{bdtt}(\text{SPARSE})$ is included in $R_{ctt}(\text{SPARSE})$ (this paper) and $R_{bctt}(\text{SPARSE})$ is included in $R_{dtt}(\text{SPARSE})$ [Ko89]. On the other hand, for any unbounded f, the classes $R_{f(n)\text{-}dtt}(\text{SPARSE})$ and $R_{f(n)\text{-}ctt}(\text{SPARSE})$ are incomparable.

From out main result, we can easily obtain further new results. For example, we show that various classes are not closed under complementation. We also obtain results that were previously known, almost directly from our main result. A typical line of reasoning is as follows: if a set is $\leq^p_{ctt}$-reducible to a sparse set, then it is $\leq^p_{ctt}$-reducible to a tally set by our result and thus its complement is $\leq^p_{dtt}$-reducible to a tally set. This complementation argument can be applied only for tally sets.

2. Preliminaries.

2.1. Notation. Let $\Sigma = \{0, 1\}$. Strings are elements of Σ^* and are denoted by lowercase letters $x, y, u, v, \ldots$. For any string x the length of a string is denoted by $|x|$. Subsets of Σ^* are denoted by capital letters $A, B, C, S, \ldots$. The set $\Sigma^* - A$ is denoted by $\overline{A}$. For a set A we use $A^{=n}(A^{\leq n})$ to denote the subset of A consisting of all strings of length $n(\leq n)$. For any set A the cardinality of A is denoted by $\|A\|$. If for all n, $\|A^{=n}\| \leq d(n)$, we say that

A is of *density* $d(n)$. We call a set S *sparse* if there exists a polynomial p such that for all n, $\|S^{\leq n}\| \leq p(n)$. A set T is called *tally* if $T \subseteq \{0\}^*$. We fix a pairing function $\lambda xy.\langle x, y\rangle$ computable in polynomial time from $\Sigma^* \times \Sigma^*$ to Σ^*. Without loss of generality we assume that for all x, y : $|x| + |y| \leq |\langle x, y\rangle| \leq 2(|x| + |y|)$. We assume that the reader is familiar with the standard Turing machine model.

2.2. Truth tables. The ordered pair $\langle\langle a_1, \ldots, a_k\rangle, \alpha\rangle$, for $k > 0$, is called a *truth-table condition of norm* k is $\langle a_1, \ldots, a_k\rangle$ is a k-tuple of strings and α is a k-ary Boolean function [LLS75]. The set $\{a_1, \ldots, a_k\}$ is called the *associated set* of the tt-condition. A function f is a *truth-table function* if f is total and $f(x)$ is a truth-table condition for every x in Σ^*. We denote the associated set of $f(x)$ by $\mathrm{Ass}(f(x))$. If, for all x, $f(x)$ has norm less than or equal to k then f is called a k-truth-table (ktt) function. We say that a tt function f is a *disjunctive* (*conjunctive*) truth-table (dtt (ctt)) function if f is a truth-table condition whose Boolean function is always a disjunction (conjunction) of its arguments

2.3. Reductions, reducibilities. Let $A_1, A_2 \subseteq \Sigma^*$. In this paper, all reductions are polynomial-time computable. We say that

1. A_1 is truth-table reducible to A_2 ($\leq^p_{tt}$-reducible) iff there exists a polynomial-time computable tt function f such that $x \in A_1$ iff $\alpha(\chi_{A_2}(a_1), \ldots, \chi_{A_2}(a_k)) =$ **true**, where $f(x)$ is $\langle\langle a_1, \ldots, a_k\rangle, \alpha\rangle$ and χ_{A_2} is the characteristic function of the set A_2.
2. A_1 is k-truth-table reducible to A_2 ($\leq^p_{k\text{-}tt}$-reducible) iff $A_1 \leq^p_{tt} A_2$ by some ktt function. A_1 is bounded-truth-table reducible to A_2 ($\leq^p_{btt}$-reducible) iff $A_1 \leq^p_{k\text{-}tt} A_2$ for some integer k.
3. A_1 is disjunctive (conjunctive) truth-table reducible ($\leq^p_{dtt}$ ($\leq^p_{ctt}$)-reducible) to A_2 iff $A_1 \leq^p_{tt} A_2$ by some dtt (ctt) function. For $k \geq 0$, A_1 is k-disjunctive (conjunctive) truth-table reducible ($\leq^p_{k\text{-}dtt}$ ($\leq^p_{k\text{-}ctt}$)) to A_2 if $A_1 \leq^p_{tt} A_2$ by some dtt (ctt) function of norm k.
4. A_1 is disjunctive (conjunctive) truth-table reducible ($\leq^p_{bdtt}$ ($\leq^p_{bctt}$)-reducible) to A_2 iff $A_1 \leq^p_{k\text{-}dtt}$ ($\leq^p_{k\text{-}ctt}$) A_2 for some integer k.
5. A_1 is positive truth-table reducible to A_2 ($\leq^p_{ptt}$-reducible) [Sel82] iff $A_1 \leq^p_{tt} A_2$ by some tt function f such that for all sets X_1, X_2, Y_1, and Y_2, if $X_1 \leq^p_{tt} X_2$ via f, $X_2 \subseteq Y_2$, and $Y_1 \leq^p_{tt} Y_2$ via f, then $X_1 \subseteq Y_1$.

We will consider languages that are reducible to sparse and tally sets. Let r be any of the above reductions. Then

$$\begin{aligned}
\text{SPARSE} &= \{S \mid S \text{ is a sparse set}\},\\
\text{co-SPARSE} &= \{S \mid \overline{S} \text{ is a sparse set}\},\\
\text{TALLY} &= \{T \mid T \text{ is a tally set}\},\\
R_r(\text{SPARSE}) &= \{A \mid A \leq^p_r S \text{ for some } S \in \text{ SPARSE}\},\\
R_r(\text{TALLY}) &= \{A \mid A \leq^p_r T \text{ for some } T \in \text{ TALLY}\}.
\end{aligned}$$

2.4. Kolmogorov complexity. The *Kolmogorov complexity* of a string x, $K(x)$, is the size of the smallest index of a Turing machine that generates x and halts. A *Kolmogorov random* string is a string x such that $K(x) \geq |x|$. For a more detailed description see, for example, [LV93].

3. Conjunctive reductions to tally sets.

THEOREM 1. $\text{SPARSE} \subseteq R_{ctt}(\text{TALLY})$.

Proof. Let S be a sparse set and let $d(n)$ a polynomial upper bound on its density, where d is a polynomial-time-computable function. Such a function d exists for every sparse set. We show that $S \in R_{ctt}(\text{TALLY})$.

We have to build a $\leq^p_{ctt}$ reduction g from S to a tally set T. We can ensure that $\mathrm{Ass}(g(x)) \cap \mathrm{Ass}(g(y)) = \emptyset$ for $|x| \neq |y|$ by building g such that every element of $\mathrm{Ass}(g(x))$ is of the

form $0^{\langle n,j\rangle}$, where n is the length of x. In the following, let $x_1, \ldots, x_{2^n}$ be the 2^n strings of length n. Note that if g is a reduction from S to T, then $x_i \in S \Leftrightarrow \text{Ass}(g(x_i)) \subseteq T$. Since this property holds for all x_i, a $\leq^p_{ctt}$ reduction generates a family of 2^n tally sets such that for all $x_i \notin S$, $\text{Ass}(g(x_i)) \not\subseteq \bigcup_{x_j \in S} \text{Ass}(g(x_j))$. Whether the reduction is possible depends on whether we can efficiently construct such a family of sets. The existence of these kinds of families has been studied in [EFF82], [EFF85], [NW88]. We will construct a family of sets $\mathcal{F} = \{Q_1, \ldots, Q_{2^n}\}$, with the following properties:

1. $Q_i \in \text{TALLY}$,
2. Q_i can be generated in polynomial time (in n),
3. For any $d(n) + 1$ sets $Q_{i_1}, \ldots, Q_{i_{d(n)}}, Q_k \in \mathcal{F}$ such that $k \notin \{i_1, \ldots, i_{d(n)}\}$, $Q_k \not\subseteq \bigcup_{j=1}^{d(n)} Q_{i_j}$.

If we set the tally set $T = \bigcup_{x_i \in S} Q_i$, then $x_i \in S$ iff $Q_i \subseteq T$, since S is of density $d(n)$. If we are able to generate Q_i in polynomial time (in n), then we can define the $\leq^p_{ctt}$ reduction f from $S^{=n}$ to T by $\text{Ass}(f(x_i)) = Q_i$. First we show by the next lemma that property 3 above follows from the following stronger property, which is easier to verify.

LEMMA 1. *Let $\mathcal{F} = \{Q_1, \ldots, Q_{2^n}\}$ be a family of sets such that for some $r > 0$, $\|Q_i\| > r \cdot d(n)$ and $\|Q_i \bigcap Q_j\| \leq r$ for $i \neq j$. Then, for any $d(n) + 1$ sets $Q_{i_1}, \ldots, Q_{i_{d(n)}}, Q_k \in \mathcal{F}$ such that $k \notin \{i_1, \ldots, i_{d(n)}\}$, $Q_k \not\subseteq \bigcup_{j=1}^{d(n)} Q_{i_j}$.*

Proof. Suppose this is not true, i.e., there exist $d(n) + 1$ sets $Q_{i_1}, \ldots, Q_{i_{d(n)}}, Q_k \in \mathcal{F}$ such that $k \notin \{i_1, \ldots, i_{d(n)}\}$ and $Q_k \subseteq \bigcup_{j=1}^{d(n)} Q_{i_j}$. Since $\|Q_k\| > r \cdot d(n)$, there must exist a j such that $1 \leq j \leq d(n)$ and $\|Q_k \bigcap Q_{i_j}\| > r$. But this contradicts the fact that the size of the intersection of any two different sets is at most r. □

One way to construct these families is as follows. Let $GF(p)$ be a finite field with a prime number of elements. Note here that we can always find a prime between x and $2x$ [Che52].

We consider polynomials over $GF(p)$ for p prime. We need an easy fact about roots of polynomials over finite fields. For more detail see §6.6 in [Coh74].

FACT 1. *Two different polynomials of degree $\leq r$ cannot intersect on more than r points in $GF(p)$.*

We represent a polynomial of degree $\leq r$ by its $r + 1$ coefficients. We view each polynomial as a $(r + 1)$-digit number in base p. With the ith polynomial, denoted by q_i, we mean the polynomial whose representation is the number base p that represents i. Consider the following family of sets: $Q_i = \{0^{\langle n,a,q_i(a)\rangle} | a \in GF(p)\}$. We will choose r and p such that the conditions of Lemma 1 are fulfilled. Observe that Q_i is a tally set of size p, and that for two different polynomials q_i and q_j, $\|Q_i \bigcup Q_j\| \leq r$. It remains to force the following requirements:

1. $p^{r+1} \geq 2^n$ (we need 2^n different sets),
2. $r \cdot d(n) < p$ (to fulfill the requirements of Lemma 1).

It is easy to verify that taking $r = \lceil \frac{2n}{\log n} \rceil$ and p the first prime larger than $r \cdot d(n)$ fulfills these two requirements.

The only thing remaining is to show that we can generate the ith set Q_i in polynomial time (in n). First we have to compute the prime number p. Since the length of the binary representation of $r \cdot d(n)$ is in $O(\log(n))$ and because there is a prime between $r \cdot d(n)$ and $2r \cdot d(n)$, we can do a brute-force search (or do a more sophisticated sieve method [Pri83]) in polynomial time. Next we have to pick the ith polynomial over $GF(p)$ (which can easily be done in polynomial time) and compute Q_i. Since p is a prime number, the operations in $GF(p)$ are simply multiplication and addition modulo p, which also can be done in polynomial time. □

Recall that the $\leq^p_{ctt}$ reduction f from $S^{=n}$ to $\bigcup_{x_i \in S} Q_i$ is defined by $\mathrm{Ass}(f(x_i)) = Q_i$. Since $\|Q_i\| = p \leq 2r \cdot d(n) \leq (4nd(n)/\log n)$, we have in fact shown that $S \in R_{O(nd(n)/\log n)-ctt}(\mathrm{TALLY})$. As shown by Saluja [Sal93], this bound is optimal.

Note that if we consider probabilistic reductions, we can randomly choose exactly one of the strings from $\mathrm{Ass}(f(x))$ and get a many-one reduction with a one-sided error. This observation is due to Schöning in [Sch93], where he shows that every sparse set many-one reduces to a tally set by a polynomial-time, randomized procedure.

COROLLARY 1. $R_{ctt}(\mathrm{SPARSE}) = R_{ctt}(\mathrm{TALLY})$.

COROLLARY 2. co-SPARSE $\subseteq R_{dtt}(\mathrm{TALLY})$.

Proof. If A is $\leq^p_{ctt}$-reducible to a tally set, then $\overline{A}$ is $\leq^p_{dtt}$-reducible to a tally set. □

The following theorem can be derived using Theorem 1. It refutes another of the conjectures from [Ko89]. (The conjecture was that $R_{bdtt}(\mathrm{SPARSE}) \not\subseteq R_{ctt}(\mathrm{SPARSE})$.)

THEOREM 2. $R_{bdtt}(\mathrm{SPARSE}) \subseteq R_{ctt}(\mathrm{TALLY})$.

Proof. Let A be $\leq^p_{k\text{-}dtt}$-reducible to some sparse set S via f. Using Theorem 1 we get that S is $\leq^p_{ctt}$-reducible to some tally set T_S via g. We will construct a tally set T and a reduction h such that $A \leq^p_{ctt} T$ via h. Define

$$T = \left\{0^{\langle n_1,\ldots,n_k\rangle} \mid n_j \in \mathbb{N} \text{ and } \exists i : 0^{n_i} \in T_S\right\}.$$

In the following it is convenient to view T as a Cartesian product. For $A_1, \ldots, A_k$ tally sets, let

$$A_1 \times \cdots \times A_k = \left\{0^{\langle n_1,\ldots,n_k\rangle} \mid 0^{n_i} \in A_i\right\}.$$

Define the $\leq^p_{ctt}$ reduction h as follows: if $f(x) = \langle\langle y_1, \ldots, y_k\rangle, \alpha\rangle$, then let $\mathrm{Ass}(h(x)) = \mathrm{Ass}(g(y_1)) \times \cdots \times \mathrm{Ass}(g(y_k))$. Note that h is polynomial-time computable since both f and g are. It remains to show that h reduces A conjunctively to T.

$$\begin{aligned}
x \in A &\Rightarrow \exists i : y_i \in S \\
&\Rightarrow \exists i : \mathrm{Ass}(g(y_i)) \subseteq T_S \\
&\Rightarrow \mathrm{Ass}(g(y_1)) \times \cdots \times \mathrm{Ass}(g(y_k)) \subseteq T. \\
x \notin A &\Rightarrow \forall i : y_i \notin S \\
&\Rightarrow \forall i \exists 0^{n_i} : 0^{n_i} \in \mathrm{Ass}(g(y_i)) \text{ and } 0^{n_i} \notin T_S \\
&\Rightarrow 0^{\langle n_1,\ldots,n_k\rangle} \notin T \\
&\Rightarrow \mathrm{Ass}(g(y_1)) \times \cdots \times \mathrm{Ass}(g(y_k)) \not\subseteq T. \qquad \square
\end{aligned}$$

Theorem 1 offers a new understanding of the class $R_{ctt}(\mathrm{SPARSE})$ and as such, it has been used in [AKM92] to prove various results.

To understand the relationship between sparse and tally sets, it is important to know which reductions are able to differentiate between tally and sparse sets and which aren't. It is well known that $R_{tt}(\mathrm{SPARSE}) = R_{tt}(\mathrm{TALLY})$ [HIS85] and our Corollary 1 gives the analog for $\leq^p_{ctt}$ reductions. On the other hand, there *do* exist reductions that are more powerful with sparse oracles than with tally oracles. This holds, for instance, for many-one reductions and for disjunctive truth-table reductions [Ko89].

As the next theorem shows, *positive* truth-table reductions on sparse and tally sets behave like $\leq^p_{ctt}$ reductions and not like $\leq^p_{dtt}$ reductions.

THEOREM 3. $R_{ptt}(\mathrm{SPARSE}) = R_{ptt}(\mathrm{TALLY})$.

The result follows immediately from the following theorem, which claims that $\leq^p_{ptt}$ reductions to tally sets capture the class $R_{tt}(\mathrm{TALLY})$.

THEOREM 4. $R_{tt}(\text{TALLY}) = R_{ptt}(\text{TALLY})$.

Proof. Let A be a set in $R_{tt}(\text{TALLY})$ and suppose T is a tally set such that $A \leq_{tt}^{p} T$ by a tt function f that is computable in time $p(n)$ where p is a polynomial. We have to show that $A \in R_{ptt}(\text{TALLY})$. We define the tally set T', which will witness the fact that $A \in R_{ptt}(\text{TALLY})$, as follows:

$$T' = \left\{0^{\langle n,0\rangle} \mid 0^n \in T\right\} \cup \left\{0^{\langle n,1\rangle} \mid 0^n \notin T\right\}.$$

We claim that $A \leq_{ptt}^{p} T'$ by the following reduction.

On input x of length n do the following:

1. If there exists an $m \leq p(n)$ such that $0^{\langle m,0\rangle}$ and $0^{\langle m,1\rangle}$ are both *not* in the oracle set, then reject;
2. else, if there exists an $m \leq p(n)$ such that $0^{\langle m,0\rangle}$ and $0^{\langle m,1\rangle}$ are both *in* the oracle set, then accept;
3. otherwise, simulate the old tt function f on input x, replacing each query 0^m by $0^{\langle m,0\rangle}$.

It is immediate that this reduction reduces A to T', since by definition of T' we are always in case 3, which implies that we just simulate f. It remains to show that the reduction is positive. Suppose for a contradiction that it isn't. Then there exist a string x of length n and two oracle sets $X \subset Y$ such that x is accepted with oracle X and rejected with oracle Y. Since x is accepted with oracle X, we cannot be in case 1, that is, it must be the case that for all $m \leq p(n)$ either $0^{\langle m,0\rangle} \in X$ or $0^{\langle m,1\rangle} \in X$. Now look at Y. If $Y \setminus X$ does not contain strings of the form $0^{\langle m,1\rangle}$ for $m \leq p(n)$, $i \in \{0, 1\}$, then $f(x)$ with oracle Y behaves in exactly the same way as $f(x)$ with oracle X. In particular, x is accepted, which contradicts our assumption. Therefore, suppose that for some $m \leq p(n)$ and $i \in \{0, 1\}$ it is the case that $0^{\langle m,i\rangle}$ occurs in Y but not in X. Then it must be the case that $0^{\langle m,|1-i|\rangle} \in X$, and therefore, since $X \subseteq Y$, both $0^{\langle m,0\rangle}$ and $0^{\langle m,1\rangle}$ are in Y. This implies that we are in case 2, and thus, x is accepted contrary to the assumption. □

Note that by the construction, it is immediate that T' is 1-tt reducible to T.

4. Conjunctive and disjunctive reductions. Gavaldà and Watanabe [GW93] showed that $R_{ctt}(\text{SPARSE}) \not\subseteq R_{dtt}(\text{SPARSE})$. Combining this result with Theorem 1, we can quickly derive the following theorem of Ko.

THEOREM 5 [Ko89]. $R_{dtt}(\text{SPARSE}) \not\subseteq R_{ctt}\text{SPARSE})$.

Proof. Let A be a set in $R_{ctt}(\text{SPARSE})$ that is not in $R_{dtt}(\text{SPARSE})$. Consider the set $\overline{A}$. Since $A \in R_{ctt}(\text{SPARSE})$ and $R_{ctt}(\text{SPARSE}) = R_{ctt}(\text{TALLY})$ by Theorem 1, we have that $A \in R_{ctt}(\text{TALLY})$. By simple complementation, it follows that $\overline{A} \in R_{dtt}(\text{TALLY})$ and therefore, $\overline{A} \in R_{dtt}(\text{SPARSE})$. Now we see that $\overline{A}$ cannot be in $R_{ctt}(\text{SPARSE})$. For suppose $\overline{A} \in R_{ctt}(\text{SPARSE})$. Then, again using Theorem 1, $\overline{A} \in R_{ctt}(\text{TALLY})$, so $A \in R_{dtt}(\text{TALLY}) \subseteq R_{dtt}(\text{SPARSE})$, contradicting our choice of A. □

Gavaldà and Watanabe's proof actually provides something stronger. They show that

$$R_{f(x)\text{-}ctt}(\text{SPARSE}) \not\subseteq R_{dtt}(\text{SPARSE})$$

for any polynomial-time-computable unbounded function f. Ko's proof of Theorem 5 does not seem to provide this generalization and the above proof does not generalize directly, because when we go conjunctively from a sparse set to a tally set, we need a polynomial number of queries. To be able to use the previous argument while keeping the number of queries small, we need a strengthening of Gavaldà and Watanabe's theorem to tally sets.

THEOREM 6. *For any polynomial-time-computable unbounded function* f, $R_{f(n)\text{-}ctt}(\text{TALLY}) \not\subseteq R_{dtt}(\text{SPARSE})$.

Proof. If we can prove the theorem for small functions f, it is immediately true for larger functions, so we may assume $f(n) \leq \log n$. For every n, let x_n be a Kolmogorov random string of length n. Define

$$A = \{ \langle 0^n, \langle i_1, b_1\rangle, \ldots, \langle i_{f(n)}, b_{f(n)}\rangle\rangle \text{ such that } 1 \leq i_1 < i_2 < \cdots < i_{f(n)} \leq n \text{ and for every } j, \text{ the } i_j\text{th bit of } x_n \text{ is } b_j\}.$$

It is immediate that $A \leq^p_{f(n)\text{-}ctt} T$, where

$$T = \left\{0^{\langle n,i,b\rangle} \mid \text{the } i\text{th bit of } x_n \text{ is } b\right\}.$$

To show that A is not $\leq^p_{dtt}$-reducible to any sparse set, leading to a contradiction, assume $A \leq^p_{dtt} S$, via reduction h, where h is n^c-time computable and $\|S^{\leq n}\| \leq n^c$.

Let A_n be the set of all strings of A of the form $\langle 0^n, \ldots\rangle$. We will show that there is a string y_n in S that is queried by many strings from A_n (Lemma 2). Suppose that a string $\langle 0^n, \langle i_1, b_1\rangle, \ldots, \langle i_{f(n)}, b_{f(n)}\rangle\rangle$ queries the string y_n. Since h is a $\leq^p_{dtt}$ reduction fro A to S and $y_n \in S$, this provides us with the $f(n)$ bits $i_1, i_2, \ldots, i_{f(n)}$ of x_n. By a careful counting argument, we show below that, for n large enough, we get enough bits of x_n from y_n to contradict the randomness of x_n.

LEMMA 2. *There exist a constant d and for every n a string y_n in S such that*

$$\|\{z \in A_n \mid y_n \in \text{Ass}(h(z))\}\| \geq n^{\frac{1}{2}f(n)-d}.$$

Proof. The number of strings in A_n is $\binom{n}{f(n)} \geq \left(\frac{n}{f(n)}\right)^{f(n)}$. Thus, for $f(n) \leq n^{\frac{1}{2}}$, $\|A_n\| \geq n^{\frac{1}{2}f(n)}$. For each string z in A_n, there is a string in $S \cap \text{Ass}(h(z))$. Since strings in A_n are certainly of length less than $2n$, the queried strings are of length at most $(2n)^c$. Thus, there are at most $((2n)^c)^c = (2n)^{c^2}$ strings of S in $\cup_{z\in A_n}\text{Ass}(h(z))$. There must be a string y_n in the set that is in $\text{Ass}(h(z))$ for at least $\|A_n\|/(2n)^{c^2}$ many z's. Since $\|A_n\| \geq n^{\frac{1}{2}f(n)}$, $\|A_n\|/(2n)^{c^2} \geq n^{\frac{1}{2}f(n)-d}$ for a suitable d. □

Given a set $Y \subseteq A_n$, let I_Y be the set of indices i_j that are mentioned in the strings from Y.

LEMMA 3. *Let $Y \subseteq A_n$; then $\|Y\| \leq \|I_Y\|^{f(n)}$.*

Proof. Each string in Y mentions exactly $f(n)$ bits of I_Y. There are exactly $\binom{\|I_Y\|}{f(n)}$ ways to select $f(n)$ bits from the set of indices I_Y, so

$$\|Y\| \leq \binom{\|I_Y\|}{f(n)} \leq \|I_Y\|^{f(n)}. \qquad \square$$

LEMMA 4. *There exists a string $y_n \in S$ such that for the set Y of strings in A_n that query y_n, $\|I_Y\| \geq n^{\frac{1}{2}-d/f(n)}$.*

Proof. Let y_n be given by Lemma 2 and let Y be the set of strings z in A_n such that $y_n \in \text{Ass}(h(z))$. Then, by Lemma 3,

$$n^{\frac{1}{2}f(n)-d} \leq \|I_Y\|^{f(n)},$$

$$\|I_Y\| \geq n^{(\frac{1}{2}f(n)-d)/f(n)} = n^{\frac{1}{2}-d/f(n)}.$$

Now, to derive a contradiction, we show how to describe x_n with fewer than n bits. To describe x_n, use the string y_n from Lemma 4. To compute y_n, we need one of the strings $z \in A_n$ that query y_n, and the index of y_n in the set of queries. The string z can be described

using $O(f(n)\log n)$ bits and the index can be described in $O(\log n)$ bits. It follows that y_n can be described using $O(f(n)\log n)$ bits. Given y_n, we can compute all the bits of x_n that are mentioned in strings form the set Y of strings in A_n that query y_n. Now look at the sequence containing all the bits of x_n that are not mentioned by Y. This requires $n - \|I_Y\| \leq n - n^{\frac{1}{2}-d/f(n)}$ bits. Since the bits described by Y all contain their index, they can be inserted into their respective position. The total number of bits needed to describe x_n is $n - n^{\frac{1}{2}-d/f(n)} + O(f(n)\log n)$, which is strictly less than n if $f(n)$ is unbounded and $\leq \log n$. □

Now we can derive the wanted theorem.

THEOREM 7. *For any polynomial-time-computable unbounded function f, $R_{f(n)\text{-}dtt}$(TALLY) $\not\subseteq R_{ctt}$(SPARSE).*

Proof. Using Theorem 6, we can use the same reasoning as in the proof of Theorem 5. Since we start from a tally set, we don't have the problem associated with the blow up in number of queries. □

The following corollaries can all be obtained from Theorems 6 and 7.

COROLLARY 3. *For any polynomial-time-computable unbounded function f, $R_{f(n)\text{-}ctt}$(SPARSE) and $R_{f(n)\text{-}dtt}$(SPARSE) are not closed under complementation.*

COROLLARY 4. *For any polynomial-time-computable unbounded function f, $R_{f(n)\text{-}ctt}$(SPARSE) and $R_{f(n)\text{-}dtt}$(SPARSE) are incomparable.*

COROLLARY 5. *For any polynomial-time-computable unbounded function f, $R_{f(n)\text{-}dtt}$(SPARSE) and $R_{f(n)\text{-}ctt}$(SPARSE) are strictly included in $R_{f(n)\text{-}tt}$(SPARSE).*

These results hold for the corresponding R_r(TALLY) classes as well. For bounded conjunctive and disjunctive reductions to sparse sets, we get the following analog.

THEOREM 8. *For all $k \geq 1$, $R_{k\text{-}ctt}$(SPARSE), $R_{k\text{-}dtt}$(SPARSE), R_{bdtt}(SPARSE), and R_{bctt}(SPARSE) are not closed under complementation, and therefore are strictly included in R_{btt}(SPARSE).*

Proof. It it not hard to see that if R_{bdtt}(SPARSE) is closed under complementation, then $R_{1\text{-}tt}$(SPARSE) $\subseteq R_{bdtt}$(SPARSE). By Theorem 2, it follows that $R_{1\text{-}tt}$(SPARSE) $\subseteq R_{ctt}$(SPARSE), contradicting [Ko89]. For the bounded conjunctive case we can argue in a similar way. □

Note that this theorem does not hold for the corresponding R_r(TALLY) classes. It follows from [Ko89] that R_m(TALLY) = $R_{k\text{-}ctt}$(TALLY) = $R_{k\text{-}dtt}$(TALLY) = R_{btt}(TALLY), and thus all these classes are closed under complementation.

Acknowledgments. We would like to thank Lance Fortnow for pointing out the relevance of [NW88] and Peter Erdös for giving us a copy of [EFF85]. We also thank Richard Gavaldà for providing us with a preliminary version of one of the proofs in [GW93] before it was available for general distribution, Peter van Emde Boas for discussing algebra with us, and two anonymous referees for helpful comments.

REFERENCES

[AHH+93] V. ARVIND, Y. HAN, L. HEMACHANDRA, J. KÖBLER, A. LOZANO, M. MUNDHENK, M. OGIWARA, U. SCHÖNING, R. SILVESTRI, AND T. THIERAUF, *Reductions to sets of low information content*, in Complexity Theory, Current Research, K. Ambos-Spies, S. Homer, and U. Schöning, eds., Cambridge University Press, London, U.K., 1993, pp. 1–45.

[AHOW92] E. ALLENDER, L. HEMACHANDRA, M. OGIWARA, AND O. WATANABE, *Relating equivalence and reducibility to sparse sets*, SIAM J. Comput., 21 (1992), pp. 521–539.

[AKM92] V. ARVIND, J. KÖBLER, AND M. MUNDHENK, *On bounded truth-table, conjunctive, and randomized reductions to sparse sets*, in Proc. 12th Conference on the Foundations of Software Technology & Theoretical Computer Science, Lecture notes in Computer Science 652, Springer-Verlag, 1992, pp. 140–151.

[BK88] R. BOOK AND K. KO, *On sets truth-table reducible to sparse sets*, SIAM J. Comput., 17 (1988), pp. 903–919.

[Che52] L. CHEBYSHEV, *Mémoire sur les nombres premiers*, Journal de Math., 17 (1852), pp. 366–390.

[Coh74] P. M. COHN, *Algebra*, Vol. 1, John Wiley & Sons, New York, 1974.

[EFF82] P. ERDÖS, P. FRANKL, AND Z. FÜREDI, *Families of finite sets in which no set is covered by the union of two others*, J. Combin. Theory Ser. A, 33 (1982), pp. 158–166.

[EFF85] P. ERDÖS, P. FRANKL, AND Z. FÜREDI, *Families of finite sets in which no set is covered by the union of r others*, Israel J. Math., 51 (1985), pp. 79–89.

[GW93] R. GAVALDÀ AND O. WATANABE, *On the computational complexity of small descriptions*, SIAM J. Comput., 22 (1993), pp. 1257–1275.

[HIS85] J. HARTMANIS, N. IMMERMAN, AND V. SEWELSON, *Sparse sets in NP-P:EXPTIME versus NEXPTIME*, Inf. Control, 65 (1985), pp. 158–181.

[HOW92] L. HEMACHANDRA, M. OGIWARA, AND O. WATANABE, *How hard are sparse sets*? in Proc. Structure in Complexity Theory, seventh annual conference, IEEE Computer Society Press, Piscataway, NJ, 1992, pp. 222–238.

[Ko89] K. KO, *Distinguishing conjunctive and disjunctive reducibilities by sparse sets*, Inform. Comp., 81 (1989), pp. 62–87.

[LLS75] R. LADNER, N. LYNCH, AND A. SELMAN, *A comparison of polynomial time reducibilities*, Theoret. Comput. Sci., 1 (1975), pp. 103–123.

[LV93] M. LI AND PAUL VITÁNYI, *An Introduction to Kolmogorov Complexity and its Applications*, Texts and Monographs in Computer Science, Springer-Verlag, Berlin, 1993.

[NW88] N. NISAN AND A. WIGDERSON, *Hardness vs. randomness*, in Proc. 29th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Piscataway, NJ, 1988, pp. 2–11.

[Pri83] P. PRITCHARD, *Prime number sieves*, J. Algorithms, 4 (1983), pp. 332–344.

[RRW94] R. RAO, J. ROTHE, AND O. WATANABE, *Upward separation for* FewP *and related classes*, Technical Report TR 488, University of Rochester, 1994. Inform. Proc. Lett., 52 (1994), pp. 175–180.

[Sal93] S. SALUJA, *Relativized limitations to left set technique and closure classes of sparse sets*, in Proc. Structure in Complexity Theory eighth annual conference, IEEE Computer Society Press, Piscataway, NJ, 1993.

[Sch93] U. SCHÖNING, *On random reductions from sparse sets to tally sets*, Inform. Proc. Lett., 46 (1993), pp. 239–241.

[Sel82] A. SELMAN, *Analogues of semirecursive sets and effective reducibilities to the study of NP complexity*, Inform. and Control, 52 (1982), pp. 36–51.

[Wat] O. WATANABE, Private communication.

SIAM J. COMPUT.
Vol. 24, No. 4, pp. 682–705, August 1995

SIZE-DEPTH TRADEOFFS FOR ALGEBRAIC FORMULAS*

NADER H. BSHOUTY†, RICHARD CLEVE†, AND WAYNE EBERLY†

Abstract. Some tradeoffs between the size and depth of algebraic formulas are shown. In particular, it is shown that, for any fixed $\epsilon > 0$, any algebraic formula of size S can be converted into an equivalent formula of depth $O(\log S)$ and size $O(S^{1+\epsilon})$. This result is an improvement over previously known results where, to obtain the same depth bound, the formula size is $\Omega(S^{\alpha})$ with $\alpha \geq 2$.

Key words. arithmetic expressions, formulas, parallel computation

AMS subject classifications. 68Q15, 68Q40

1. Introduction. A classical result of Brent [1] implies that for any algebraic formula there is an algebraic circuit of "small" depth and "similar" size that computes the same function; see also Muller and Preparata [10]. More precisely, if the formula has size S then the circuit has depth $O(\log S)$ and size $O(S)$. This result holds for formulas over any field. We believe that a natural question to consider is whether for any algebraic formula there is an equivalent *formula* of small depth and similar size.

Since any circuit of depth $O(\log S)$ can be transformed into a formula of the same depth with size polynomial in S, it follows immediately from Brent's result that there is also a formula of depth $O(\log S)$ and size $S^{O(1)}$ that computes the same function as the original formula of size S. Applying this to the specific circuits that result from Brent's construction yields formulas with size as large as $\Omega(S^{\alpha})$ with $\alpha \geq 2$. Simple changes in Brent's construction may improve the exponent, but straightforward modifications do not appear to result in exponents arbitrarily close to one.

A widely investigated problem that is related to Brent's result, as well as our work, is the "formula evaluation problem," where the goal is to construct a "universal formula evaluator" algorithm. Such an algorithm takes as input a description of a formula with all of its inputs specified and produces as output the value of the formula. Parallel algorithms for this problem have been proposed by Gupta [6], Miller and Reif [9], Buss [2], Buss et al. [3], and Kosaraju and Delcher [8]. These yield *NC* algorithms for the problem that also produce, for any given formula of size S, a *circuit* of depth $O(\log S)$. When these circuits are expressed as formulas, the sizes are $\Omega(S^{\alpha})$ for various $\alpha \geq 2$. In the case of division-free formulas, the exponents are smaller, but nevertheless bounded above one. As an example, in §8 we exhibit division-free formulas for which Miller and Reif's method produces formulas with such a polynomial size blowup.

In this paper, we show that over any field $\mathcal{F}$, for any fixed $\epsilon > 0$, for any formula of size S with operations from $\{+, -, \times, \div\} \cup \mathcal{F}$, there are equivalent formulas with

1. depth $O(\log S)$ and size $O(S^{1+\epsilon})$,
2. depth $O(\log^{1+\epsilon} S)$ and size $S^{1+O(1/\log\log S)}$,
3. depth $O(S^{\epsilon})$ and size $O(S)$.

For the third of the above results, when the field size is less than S and the formula contains divisions ($\div$), the method that we use adds new variables to the formula. The

*Received by the editors June 9, 1992; accepted for publication (in revised form) January 26, 1994. A preliminary version of this paper was presented at the 32nd Annual Institute of Electrical and Electronics Engineers Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1991. This research was supported in part by the National Sciences and Engineering Research Council of Canada.

†Department of Computer Science, University of Calgary, Calgary, Alberta, Canada T2N 1N4 (bshouty@cpsc.ucalgary.ca), (cleve@cpsc.ucalgary.ca), and (eberly@cpsc.ucalgary.ca).

formula is constant with respect to these new variables, but we are unable to eliminate them—while preserving the size and depth bounds—without the risk of introducing a division by zero.

Also, for Boolean formulas with operations from $\{\wedge, \vee, \neg\}$, we obtain results similar to those above.

The techniques that we use include a multilevel extension of Brent's tree-decomposition method as well as other restructuring methods.

The organization of the remainder of this paper is as follows. Section 2 contains basic definitions and notation. Brent's construction is reviewed in §3, and our multilevel extension of his tree-decomposition method is given in §4. This is applied to obtain size–depth tradeoffs for division-free formulas in §5 (Boolean formulas can be regarded as a special case of division-free arithmetic formulas). In §6, this is extended to general formulas with division. We treat the division-free case separately before considering the general case because the constants that we obtain in the division-free case are better and, in the division-free case, parts of our construction are much simpler. This permits a clearer, more gradual presentation of the different techniques that apply in the general case. Section 7 contains size–depth tradeoffs for simple formulas. Section 8 describes some specific formulas that appear to exhibit increases in size when their depth is reduced and some known lower bounds on the size–depth tradeoff due to Commentz-Walter [4] and Commentz-Walter and Sattler [5].

2. Definitions and notation. For a field $\mathcal{F}$, a *formula* over $(\mathcal{F}, +, -, \times, \div)$ of depth d is defined as follows. A depth 0 formula is either c for some $c \in \mathcal{F}$ (a *constant*), or x_u for some $u \in \{1, 2, \ldots\}$ (an *input*). For $d > 0$, a depth d formula is $(F * G)$, where $* \in \{+, -, \times, \div\}$, F and G are formulas of depth d_F and d_G, respectively, and $d = \max(d_F, d_G) + 1$. The *size* of a formula F, denoted by $|F|$, is, informally, the number of occurrences of inputs and constants in the formula. More formally, a depth 0 formula has size one, and $|(F * G)| = |F| + |G|$ $(* \in \{+, -, \times, \div\})$.

A formula over $(\mathcal{F}, +, -, \times, \div)$ corresponds to a rational function in $\mathcal{F}(x_1, \ldots, x_n)$ (for some n) in a natural way, provided that it does not involve a division by a formula equivalent to zero. For formulas F and G, $F \equiv G$ denotes that they correspond to the same rational function. Hence $\equiv$ denotes equivalence in the function semantics sense.

A *division-free formula* is one that has no divisions. Clearly, division-free formulas correspond to polynomials.

A *simple formula* is one that is division-free and for which at least one argument of each multiplication operation is either an input or a constant. Thus, a depth 0 simple formula is either a constant or an input, and for depth $d > 0$ a depth d simple formula is $(F * G)$, where F and G are simple formulas, $* \in \{+, -, \times\}$, and if $* = \times$ then either F or G has depth 0.

In order to denote decompositions of a formula, we define an *extended formula*, which is allowed to take *auxiliary inputs*, which are input symbols that are not from $\{x_1, x_2, \ldots\}$. For clarity, in extended formulas we write all auxiliary inputs as "arguments" to the formula. For example, the extended formula $F(y)$ has auxiliary input symbol y. If G is a formula, then $F(G)$ denotes the formula $F(y)$ modified by substituting G for the symbol y.

The *size* of an extended formula is defined recursively as above, except that auxiliary inputs are not counted (that is, an auxiliary input has size 0). The *depth* of an extended formula is also defined recursively as above, except auxiliary inputs *are* included in this definition.

We use special terminology to denote the number of occurrences of auxiliary variables in extended formulas and the depth of particular auxiliary variables in extended formulas. For $\mathcal{A} \subseteq \{y_1, \ldots, y_m\}$, $|G(y_1, \ldots, y_m)|_{\mathcal{A}}$ denotes the total number of occurrences of inputs from $\mathcal{A}$ in $G(y_1, \ldots, y_m)$. Also, for $\mathcal{A} \subseteq \{y_1, \ldots, y_m\}$, $|G(y_1, \ldots, y_m)|_{+\mathcal{A}}$ denotes the total number of occurrences of inputs from $\mathcal{F} \cup \{x_1, x_2, \ldots\} \cup \mathcal{A}$ in $G(y_1, \ldots, y_m)$. In particular,

an extended formula $G(y_1, \ldots, y_m)$ is *read-once* with respect to an auxiliary input y_i if and only if $|G(y_1, \ldots, y_m)|_{\{y_i\}} = 1$. Note that for formulas $F_1, \ldots, F_m$,

$$|G(F_1, \ldots, F_m)| = |G(y_1, \ldots, y_m)| + \sum_{i=1}^{m} |F_i| \cdot |G(y_1, \ldots, y_m)|_{\{y_i\}}.$$

For $\mathcal{A} \subseteq \{y_1, \ldots, y_m\}$, $\text{depth}_{\mathcal{A}}(G(y_1, \ldots, y_m))$ denotes the maximum depth of any variable from $\mathcal{A}$ in $G(y_1, \ldots, y_m)$.

As usual, for $k \geq 0$, $\{0, 1\}^k$ denotes all binary strings of length k. Furthermore, ε denotes the empty string and $\{0, 1\}^{\leq k}$ denotes all binary strings of length less than or equal to k.

3. Brent's construction. Brent's result is partially based on the following lemma, which concerns ways of partitioning trees into pieces of various sizes.

LEMMA 3.1 (Brent [1]). *For any formula F and any m such that $1 < m \leq |F|$, there exist an extended formula $G(y)$ that is read-once with respect to y, formulas U and V, and an operation $*$ such that*

(i) $F = G(U * V)$,

(ii) $|G(y)| \leq |F| - m$ *and* $|U|, |V| < m$.

For a formula F of size greater than or equal to 5, Brent applies Lemma 3.1 with $m = \lceil \frac{1}{2}(|F| + 1) \rceil$, thereby "decomposing" F into three pieces $G(y)$, U, and V, each of size at most $\lceil \frac{1}{2}(|F| + 1) \rceil$. Then, using a recursive technique, he translates $G(y)$ into a *circuit* of size $O(|G(y)|)$ and depth $O(\log |G(y)|)$ that computes A, B, C, and D such that

$$G(y) \equiv \frac{(A \times y) + B}{(C \times y) + D};$$

he translates U into an equivalent *circuit* $\hat{U}$ of size $O(|U|)$ and depth $O(\log |U|)$, and similarly translates V into $\hat{V}$.

Finally, Brent expresses F as the required circuit by the identity

$$F \equiv \frac{(A \times (\hat{U} * \hat{V})) + B}{(C \times (\hat{U} * \hat{V})) + D}.$$

This construction cannot be expected to produce a logarithmic-depth, linear-size *formula* equivalent to F for a number of reasons. Firstly, since the components $\hat{U}$ and $\hat{V}$ each appear twice in the above expression and may each be of size approximately $\frac{1}{2}|F|$, at each recursive step in the construction the formula size could (approximately) double. Since the recursion may require approximately $\log |F|$ steps to bottom out, the logarithmic-depth formula that results could have size at least quadratic in $|F|$. Secondly, the minimum possible sum of the sizes of the formulas for A, B, C, and D may be at least double that of the formula size for $G(y)$, which also results in a size doubling at each recursive step, and thus a quadratic blowup in the final size. It should be noted that this second observation applies even in the case of division-free formulas, where it may be assumed that $C = 0$ and $D = 1$.

4. Partitioning formulas. The above derivation suggests that Brent's construction can result in at least a quadratic-size blowup when used to produce a formula instead of a circuit, because it is applied recursively to subformulas that are almost half as large as the original formula. In order to reduce this blowup in size we will use a partitioning method that produces much smaller subformulas. In particular, given a formula F and parameter $k \geq 2$ such that $|F| > k$, this construction produces subformulas of size at most $\lfloor \frac{1}{k}|F| + 1 \rfloor$. The construction produces a set of "interior" extended formulas and operations corresponding to interior nodes

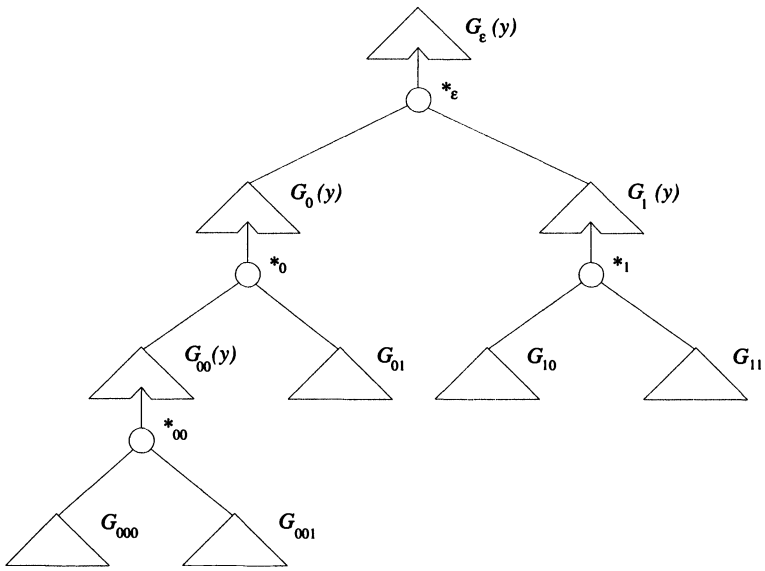


FIG. 1. *Example of a decomposition.*

of a binary tree, and "boundary" formulas corresponding to leaves, which can be combined to produce the input formula F.

Suppose, for example, that $k = 4$. Then the construction might produce a decomposition of F of the form shown in Fig. 1.

The decomposition includes interior formulas and operations $G_\varepsilon(y)$, $*_\varepsilon$, $G_0(y)$, $*_0$, $G_{00}(y)$, $*_{00}$, $G_1(y)$, $*_1$, and boundary formulas G_{000}, G_{001}, G_{01}, G_{10}, G_{11} such that each interior formula has size at most $\lfloor \frac{1}{4}|F| \rfloor$, each boundary formula has size at most $\lfloor \frac{1}{4}|F| + 1 \rfloor$, and

$$F = G_\varepsilon(G_0(G_{00}(G_{000} *_{00} G_{001}) *_0 G_{01}) *_\varepsilon G_1(G_{10} *_1 G_{11})).$$

As in this example, each subformula produced by this construction will have a subscript in $\{0, 1\}^*$ indicating a path from the "root" formula $G_\varepsilon(y)$ to that subformula in the tree comprising the decomposition of F. Two sets of subscripts, *Interior* and *Boundary*, are associated with any decomposition. *Interior* is the set of subscripts of interior extended formulas and operations, and *Boundary* is the set of subscripts of the leaf formulas in the decomposition.

Using this notation, *Interior* $= \{\varepsilon, 0, 00, 1\}$ and *Boundary* $= \{000, 001, 01, 10, 11\}$ in the above example. In contrast, Brent's construction always produces one interior formula $G_\varepsilon(y)$ and operation $*_\varepsilon$, and two leaf formulas $G_0(y)$ and $G_1(y)$ so that *Interior* $= \{\varepsilon\}$ and *Boundary* $= \{0, 1\}$.

Lemma 4.1, then, is a multilevel version of Brent's partitioning lemma. Informally, it states that every formula F can be partitioned into $4k - 3$ or fewer pieces, each of size at most $\lfloor \frac{1}{k}|F| + 1 \rfloor$. Parts (i)–(v) establish that the decomposition is well defined and correct, while parts (vi)–(viii) establish that it is small.

LEMMA 4.1. *For any formula F and any positive integer k such that $2 \leq k < |F|$, there exist finite sets of indices Interior, Boundary $\subset \{0, 1\}^*$, extended formulas $G_\alpha(y)$ and operations $*_\alpha$ for all $\alpha \in$ Interior, and formulas G_β for all $\beta \in$ Boundary such that these form a well-defined decomposition of F as follows:*

(i) $\varepsilon \in$ *Interior*.
(ii) *Interior* $\cap$ *Boundary* $= \emptyset$.
(iii) *For all* $\alpha \in$ *Interior, both* $\alpha 0$ *and* $\alpha 1$ *are in Interior* $\cup$ *Boundary*.
(iv) *For all* $\beta \in$ *Boundary, neither* $\beta 0$ *nor* $\beta 1$ *is in Interior* $\cup$ *Boundary*.
(v) *If* U_γ *is as follows for all* $\gamma \in$ *Interior* $\cup$ *Boundary, then* $F = U_\varepsilon$:

$$U_\gamma = \begin{cases} G_\gamma(U_{\gamma 0} *_\gamma U_{\gamma 1}) & \text{if } \gamma \in \textit{Interior}, \\ G_\gamma & \text{if } \gamma \in \textit{Boundary}. \end{cases}$$

(vi) $|G_\alpha(y)| \leq \lfloor \frac{1}{k}|F| \rfloor$ *and* $|G_\beta| \leq \lfloor \frac{1}{k}|F| + 1 \rfloor$ *for all* $\alpha \in$ *Interior*, $\beta \in$ *Boundary*.
(vii) *Interior* $\subseteq \{0, 1\}^{\leq k-2}$ *and* $|\textit{Interior}| \leq 2k - 2$.
(viii) *Boundary* $\subseteq \{0, 1\}^{\leq k-1}$ *and* $|\textit{Boundary}| \leq 2k - 1$.

Proof. Let F be an arbitrary formula and let k be an integer such that $2 \leq k < |F|$. We shall first give a construction for the sets *Interior* and *Boundary*, the extended formulas $G_\alpha(y)$ and operations $*_\alpha$ for $\alpha \in$ *Interior*, and formulas G_β for $\beta \in$ *Boundary*, and demonstrate that these give a well-defined decomposition of F.

To begin, set $U_\varepsilon = F$ so that $|U_\varepsilon| = |F| = |F| - |\varepsilon| \cdot \lfloor \frac{1}{k}|F| + 1 \rfloor$. Initially, *Interior* and *Boundary* are empty. We will use a third set, *Undecided*, as well; throughout the construction *Undecided* will contain all subscripts $\gamma \in \{0, 1\}^*$ such that U_γ has been defined but γ has not been added yet to either *Interior* or *Boundary*. Thus *Undecided* is initially set to $\{\varepsilon\}$.

To continue, let $\gamma \in$ *Undecided*; then U_γ has been defined, the size of U_γ is at most $|F| - |\gamma| \cdot \lfloor \frac{1}{k}|F| + 1 \rfloor$, and $\gamma \notin$ *Interior* $\cup$ *Boundary*. If $|U_\gamma| \leq \lfloor \frac{1}{k}|F| + 1 \rfloor$ then γ is added to *Boundary* and removed from *Undecided*, and G_γ is set to be U_γ. Otherwise, γ is moved from *Undecided* to *Interior* and Lemma 3.1 is applied to U_γ with $m = |U_\gamma| - \lfloor \frac{1}{k}|F| \rfloor$ to define an extended formula $G_\gamma(y)$ that is read-once with respect to y, an operation $*_\gamma$, and formulas $U_{\gamma 0}$, $U_{\gamma 1}$ such that

1. $U_\gamma = G_\gamma(U_{\gamma 0} *_\gamma U_{\gamma 1})$;
2. $|G_\gamma(y)| \leq |U_\gamma| - m = \lfloor \frac{1}{k}|F| \rfloor$;
3. $|U_{\gamma 0}|, |U_{\gamma 1}| < m$.

Note that $|U_{\gamma 0}| \leq |F| - |\gamma 0| \cdot \lfloor \frac{1}{k}|F| + 1 \rfloor$ and $|U_{\gamma 1}| \leq |F| - |\gamma 1| \cdot \lfloor \frac{1}{k}|F| + 1 \rfloor$. The subscripts $\gamma 0$ and $\gamma 1$ are now added to *Undecided*.

The construction terminates when *Undecided* $= \emptyset$.

To establish (i)–(vi), suppose the construction eventually terminates.

Initially, *Undecided* $= \{\varepsilon\}$. Clearly, every subscript that is ever included in the set *Undecided* is eventually moved to either *Interior* or *Boundary*. Since $|F| > k \geq 2$, $|U_\varepsilon| = |F| > \lfloor \frac{1}{2}|F| + 1 \rfloor \geq \lfloor \frac{1}{k}|F| + 1 \rfloor$ and ε is not added to *Boundary*. Thus part (i) follows.

It is easily established by induction on $|\gamma|$ that no string γ is ever added to *Undecided* more than once during this construction. Since every subscript that is removed from *Undecided* is added to exactly one of *Interior* or *Boundary*, part (ii) is correct.

Part (iii) follows from the fact that, for any string α, the strings $\alpha 0$ and $\alpha 1$ are both added to *Undecided* when α is moved from *Undecided* to *Interior*.

Part (iv) follows from the fact that $\beta 0$ and $\beta 1$ can only be added to *Undecided* if β is included in *Interior*, and from part (ii).

Part (v) is easily established by induction on $|\gamma|$.

Part (vi) is a consequence of the condition used to decide whether to add a string β to *Boundary* and of the correctness of Brent's partitioning method.

Thus, properties (i)–(vi) are established if the construction terminates. To see that it does, recall that no subscript γ is ever included in *Undecided* more than once and consider $\gamma \in \{0, 1\}^*$ such that γ is added to *Undecided* during the construction. Clearly, $0 < |U_\gamma| \leq |F| - |\gamma| \cdot \lfloor \frac{1}{k}|F| + 1 \rfloor$, so $|\gamma| < k$. Therefore, the construction does eventually halt, defining

sets *Boundary* $\subseteq \{0,1\}^{\leq k-1}$ and *Interior* $\subseteq \{0,1\}^{\leq k-2}$ and extended formulas, operations, and formulas such that properties (i)–(vi) hold.

In order to establish parts (vii) and (viii), define $Interior_\gamma \subseteq Interior$ for $\gamma \in Interior$ as

$$Interior_\gamma = Interior \cap \{\delta \in \{0,1\}^* : \gamma \text{ is a prefix of } \delta\}$$

with γ considered here to be a prefix of itself. Then it can be established by induction on $k - |\gamma|$ that

$$|Interior_\gamma| \leq \left\lceil \frac{2 \cdot |U_\gamma| \cdot k}{|F|} \right\rceil - 2$$

for all $\gamma \in Interior$ as follows: If $\gamma \in Interior$ and both $\gamma 0$, $\gamma 1$ are in *Boundary*, then

$$|Interior_\gamma| = 1 \leq \left\lceil \frac{2 \cdot |U_\gamma| \cdot k}{|F|} \right\rceil - 2,$$

since $|U_\gamma| > \frac{|F|}{k}$. If $\gamma \in Interior$ and exactly one (say, $\gamma 0$) of $\gamma 0$ and $\gamma 1$ is in *Interior*, then

$$|Interior_\gamma| = |Interior_{\gamma 0}| + 1 \leq \left\lceil \frac{2 \cdot |U_{\gamma 0}| \cdot k}{|F|} \right\rceil - 1 \leq \left\lceil \frac{2 \cdot |U_\gamma| \cdot k}{|F|} \right\rceil - 2,$$

since $|U_\gamma| - |U_{\gamma 0}| \geq \frac{|F|}{k}$. Finally, if γ, $\gamma 0$, and $\gamma 1$ are all in *Interior* then, since $|U_\gamma| \geq |U_{\gamma 0}| + |U_{\gamma 1}|$,

$$\begin{aligned}
|Interior_\gamma| &= |Interior_{\gamma 0}| + |Interior_{\gamma 1}| + 1 \\
&\leq \left\lceil \frac{2 \cdot |U_{\gamma 0}| \cdot k}{|F|} \right\rceil + \left\lceil \frac{2 \cdot |U_{\gamma 1}| \cdot k}{|F|} \right\rceil - 3 \\
&\leq \left(\left\lceil \frac{2 \cdot |U_{\gamma 0}| \cdot k}{|F|} + \frac{2 \cdot |U_{\gamma 1}| \cdot k}{|F|} \right\rceil + 1 \right) - 3 \\
&\leq \left\lceil \frac{2 \cdot |U_\gamma| \cdot k}{|F|} \right\rceil - 2
\end{aligned}$$

as desired.

Therefore, since $F = U_\varepsilon$ and $Interior = Interior_\varepsilon$, the size of *Interior* is at most $\lceil \frac{2 \cdot |F| \cdot k}{|F|} \rceil - 2 = 2k - 2$, which is sufficient to establish property (vii). Property (viii) also follows because elements of the sets *Interior* and *Boundary* correspond, respectively, to the internal nodes and leaves of a binary tree, so $|Boundary| = |Interior| + 1$. □

5. Division-free formulas. In this section, we show how to apply Lemma 4.1 to balance any division-free formula. In the general case there are additional complications that do not occur in the division-free case. These additional complications are related to the possibility of introducing a division by zero when a constant is substituted in an extended variable of a subformula. Techniques for avoiding this problem are explained in §6.

It should be noted that the results in this section can be easily adapted to the problem of balancing Boolean formulas. This is true for two reasons. Firstly, over the basis $\{\wedge, \oplus, 1\}$, Boolean formulas are equivalent to division-free arithmetic formulas over the two element field. Secondly, Boolean formulas over the more conventional Boolean basis $\{\wedge, \vee, \neg\}$ can be converted into formulas over the other basis by the identities

$$F \vee G \equiv \neg(\neg F \wedge \neg G) \qquad \text{and} \qquad \neg F \equiv 1 \oplus F.$$

Clearly, this conversion only increases the formula size by a linear factor, so the results about division-free arithmetic formulas apply to Boolean formulas within a linear factor.

Informally, Lemma 4.1 shows that a formula can be suitably "partitioned" into "small" subformulas. Lemma 5.1 establishes that each interior piece (an extended subformula) in this partition can be restructured so that its depth with respect to its auxiliary variable is bounded by a small constant without incurring a "large" increase in the total size and depth of the extended subformula.

LEMMA 5.1. *For any division-free extended formula $G(y)$ that is read-once with respect to y, there exists an extended formula $H(y, a, b)$, and formulas A and B such that*

(i) $G(y) \equiv H(y, A, B)$;
(ii) $|A|, |B| \leq |G(y)| + 1$;
(iii) $|H(y, a, b)|_{\{y\}} = 1$ (*i.e., $H(y, a, b)$ is read-once with respect to y*);
(iv) $|H(y, a, b)|_{\{a,b\}} \leq 2$;
(v) $|H(y, a, b)| = 0$ (*i.e., $H(y, a, b)$ has no variables other than y, a, and b*);
(vi) $\text{depth}_{\{y\}}(H(y, a, b)) = 2$;
(vii) $\text{depth}_{\{a,b\}}(H(y, a, b)) = 2$.

Proof. Since $G(y)$ is division-free and read-once with respect to y, there exist formulas P and Q such that

$$\begin{aligned} G(y) &\equiv (P \times y) + Q \\ &= H(y, P, Q) \end{aligned}$$

for $H(y, a, b) = (a \times y) + b$. Clearly, $H(y, a, b)$ satisfies claims (iii)–(vii) so it suffices to show that there exist formulas A and B satisfying claims (i) and (ii) to complete the proof. The remaining claims follow by induction on the depth d of the variable y in $G(y)$. When $d = 0$, we must have $G(y) = y$, so setting $A = 1$ and $B = 0$ satisfies (i) and (ii). If $d > 0$ then either

1. $G(y) \equiv G_1(y) + G_2$,
2. $G(y) \equiv G_1(y) - G_2$,
3. $G(y) \equiv G_2 - G_1(y)$, or
4. $G(y) \equiv G_1(y) \times G_2$,

where $G_1(y)$ is a division-free extended formula that is read-once with respect to y such that the depth of y in $G_1(y)$ is $d - 1$ and G_2 is a division-free formula. Also, $|G(y)| = |G_1(y)| + |G_2|$. By the inductive hypothesis,

$$G_1(y) \equiv (A_1 \times y) + B_1,$$

where A_1 and B_1 are division-free formulas such that $|A_1|, |B_1| \leq |G_1(y)| + 1$. Now, in case 1 we can set $A = A_1$ and $B = B_1 + G_2$; in case 2, $A = A_1$ and $B = B_1 - G_2$; in case 3, $A = -1 \times A_1$ and $B = G_2 - B_1$; and in case 4, $A = A_1 \times G_2$ and $B = B_1 \times G_2$. In each case, $G(y) \equiv H(y, A, B)$,

$$|A| \leq |G_2| + |A_1| \leq |G_2| + (|G_1(y)| + 1) \leq |G(y)| + 1,$$

and

$$|B| = |B_1| + |G_2| \leq (|G_1(y)| + 1) + |G_2| = |G(y)| + 1$$

as required. □

The preceding lemmas will be applied as follows: Think of k as a large but fixed integer. Now, consider any division-free formula F of size S. Let the goal be to restructure F so that

its depth is $O(\log S)$ while maintaining a size that is $O(S^{1+1/\log k})$. If $S \leq k$ (or a constant multiple of k) then the depth is constant because k is fixed, so no restructuring is necessary. Otherwise, Lemma 4.1 can be applied to F yielding a decomposition of F into pieces $G_\alpha(y)$ and $*_\alpha$ for $\alpha \in$ *Interior*, and G_β for $\beta \in$ *Boundary*. Lemma 5.1 can be applied to each $G_\alpha(y)$ yielding formulas $H_\alpha(y, a, b)$, A_α, and B_α with the properties described in the lemma. Then F can be "reassembled" with $H_\alpha(y, A_\alpha, B_\alpha)$ in place of each $G_\alpha(y)$. The result is a modified formula that is equivalent to F.

For example, consider again the formula shown in Fig. 1. Applying Lemma 5.1 to $G_\alpha(y)$ for each $\alpha \in$ *Interior*, we obtain formulas $H_\alpha(y, a, b)$, A_α, and B_α that can be assembled to obtain the formula equivalent to F that is shown in Fig. 2.

In general, this process will roughly double the size of the formula, and the depth of the modified formula will be approximately $3k$ plus the maximum of the depth of the "residual" subformulas A_α and B_α for $\alpha \in$ *Interior* and G_β for $\beta \in$ *Boundary*. Furthermore, by Lemma 4.1 the sizes of these residual subformulas are all roughly $\frac{1}{k}S$.

Now, consider what happens when all of the above steps are applied recursively to all of the residual subformulas of size more than some constant multiple of k. Since each recursive step reduces the size of the residuals by approximately a factor of $\frac{1}{k}$, the recursion will bottom out in approximately $\log_k S$ steps. Thus, the final depth is approximately $3k \log_k S + O(k) \in O(\log S)$ and the final size is approximately

$$2^{\log_k S} S \in O(S^{1+1/\log k})$$

as required.

A more precise exposition of the above yields the following theorem.

THEOREM 5.2. *For any division-free formula F of size S and any integer $k \geq 2$, there exists a formula G such that*

(i) $G \equiv F$,

(ii) $\text{depth}(G) \leq (3(k-1)/\log k) \log S + (3k-1)$,

(iii) $|G| \leq S^{1+1/\log k} + 4(k-1)(S^{1/\log k} - 1)$.

Proof. We shall first define a *basic transformation* of a formula F of size S that yields a restructured formula that is equivalent to F. The restructured formula shall be presented in separate pieces as follows:

1. an *inner* portion $E(w_\delta)_{\delta \in \Delta}$, which is an extended formula over some set of auxiliary variables w_δ (where δ ranges over some index set Δ);

2. an *outer* portion, which consists of a formula W_δ for each auxiliary variable w_δ of the inner portion.

Such a representation of a formula is denoted by the three-tuple

$$\langle \Delta, E(w_\delta)_{\delta \in \Delta}, \{W_\delta : \delta \in \Delta\} \rangle.$$

It is always understood that this represents the formula $E(W_\delta)_{\delta \in \Delta}$.

If $S \leq 3k$ then no restructuring is necessary, so we set $\Delta = \{\varepsilon\}$, $E(w_\varepsilon) = w_\varepsilon$, and $W_\varepsilon = F$.

If $S > 3k$ then we first apply Lemma 4.1 to F and k yielding sets *Interior*, *Boundary*, extended formulas $G_\alpha(y)$ and operations $*_\alpha$ for each $\alpha \in$ *Interior*, and formulas G_β for each $\beta \in$ *Boundary* satisfying all the conditions of the lemma. Secondly, we apply Lemma 5.1 to each $G_\alpha(y)$ yielding $H_\alpha(y, a, b)$, A_α, and B_α satisfying the conditions of this lemma.

Now, define the inner portion $E(w_\delta)_{\delta \in \Delta}$ as follows: for each element γ of *Interior* or *Boundary*, define an extended formula V_γ (where it will be understood that the auxiliary

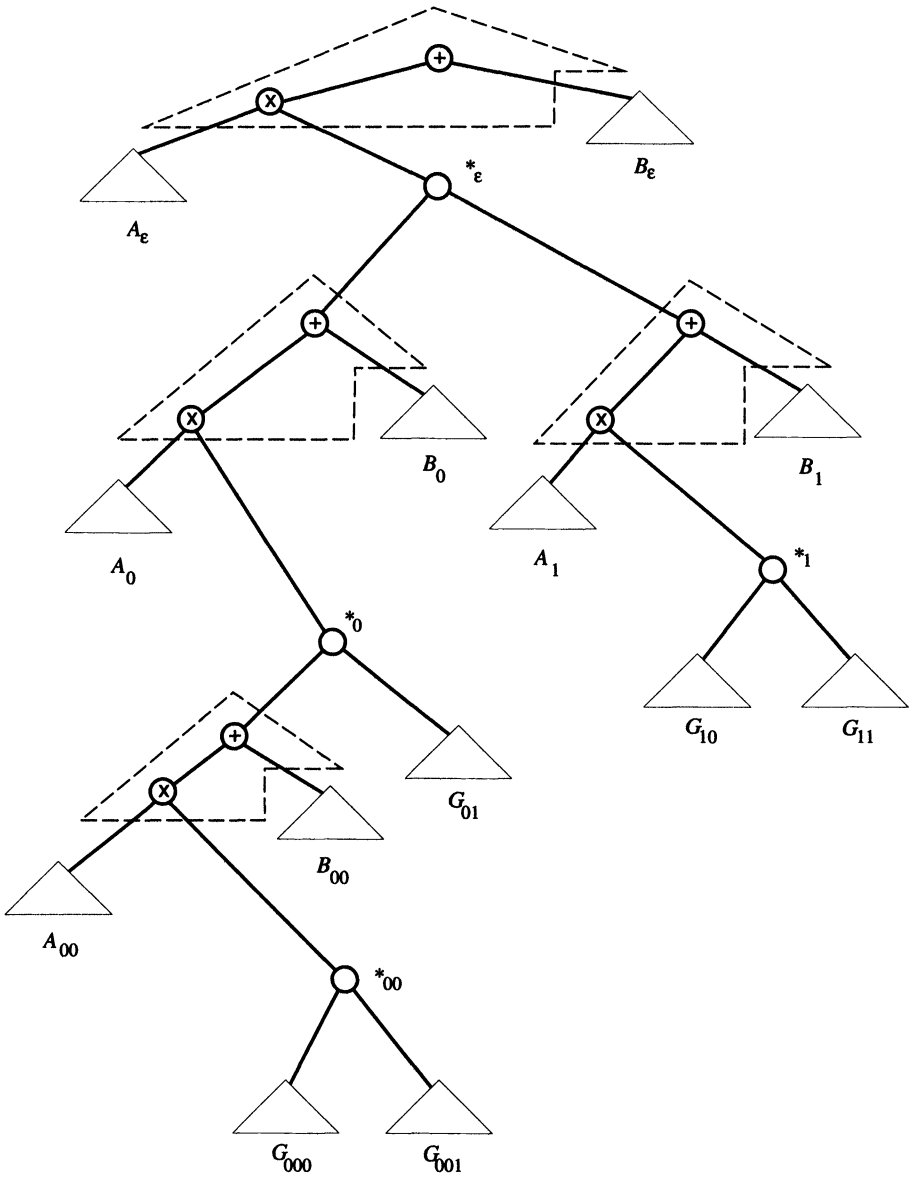


FIG. 2. *Result of basic transformation.*

variables are a_α and b_α for each $\alpha \in$ *Interior*, and g_β for each $\beta \in$ *Boundary*) by:

$$V_\gamma = \begin{cases} H_\gamma(V_{\gamma 0} *_\gamma V_{\gamma 1}, a_\gamma, b_\gamma) & \text{if } \gamma \in \textit{Interior}, \\ g_\gamma & \text{if } \gamma \in \textit{Boundary}. \end{cases}$$

Then set the inner portion as V_ε.

For each $\alpha \in$ *Interior*, set the outer formulas corresponding to auxiliary variables a_α and b_α as A_α and B_α, respectively, and for each $\beta \in$ *Boundary*, set the outer formula corresponding to g_β as G_β.

To simplify the notation at later stages, the auxiliary variables of the inner portion are renamed as w_δ, where δ ranges over some index set Δ (so $E(w_\delta)_{\delta \in \Delta} = V_\varepsilon$), and the cor-

responding outer formulas are renamed as W_δ. This completes the definition of the basic transformation of F.

We now investigate properties of the basic transformation. Clearly, $E(W_\delta)_{\delta\in\Delta} \equiv F$ since $H_\alpha(y, A_\alpha, B_\alpha) \equiv G_\alpha(y)$ for all $\alpha \in$ *Interior*. Also, using the fact that $\text{depth}_{\{y\}}(H_\alpha(y, a, b)) = 2$ and $\text{depth}_{\{a,b\}}(H_\alpha(y, a, b)) = 2$ for each $\alpha \in$ *Interior* and *Interior* $\subseteq \{0, 1\}^{\leq k-2}$, it is straightforward to verify that

$$\text{depth}(E(w_\delta)_{\delta\in\Delta}) \leq 3(k-1).$$

We now examine the size of $E(W_\delta)_{\delta\in\Delta}$. When F is transformed to $E(W_\delta)_{\delta\in\Delta}$, any size increase is a result of the replacement of $G_\alpha(y)$ by $H_\alpha(y, A_\alpha, B_\alpha)$ for $\alpha \in$ *Interior*. Since $|A_\alpha|, |B_\alpha| \leq |G_\alpha(y)|+1$ and $|H_\alpha(y, a, b)|_{\{a,b\}} \leq 2$, there are at most two instances of variables or constants in $H_\alpha(y, A_\alpha, B_\alpha)$ corresponding to each variable in $G_\alpha(y)$. In addition, there may be two additional variables or constants for each $\alpha \in$ *Interior*. Since $|$*Interior*$| \leq 2k-2$,

$$|E(W_\delta)_{\delta\in\Delta}| \leq 2S + 2(2k-2) = 2S + 4(k-1).$$

Finally, note that since $|A_\alpha|, |B_\alpha| \leq \lfloor \frac{1}{k} S \rfloor + 1 \leq \frac{1}{k} S + 1$ for each $\alpha \in$ *Interior* and $|G_\beta| \leq \lfloor \frac{1}{k} S + 1 \rfloor \leq \frac{1}{k} S + 1$ for each $\beta \in$ *Boundary*, it follows that, for all $\delta \in \Delta$,

$$|W_\delta| \leq \tfrac{1}{k} S + 1.$$

Thus, in summary, the basic transformation converts a formula F of size S to $\langle \Delta, E(w_\delta)_{\delta\in\Delta}, \{W_\delta : \delta \in \Delta\}\rangle$, where

1. $E(W_\delta)_{\delta\in\Delta} \equiv F$;

2. $\text{depth}(E(w_\delta)_{\delta\in\Delta}) \leq \begin{cases} 3(k-1) & \text{if } S > 3k, \\ 0 & \text{if } S \leq 3k; \end{cases}$

3. $|E(W_\delta)_{\delta\in\Delta}| \leq \begin{cases} 2S + 4(k-1) & \text{if } S > 3k, \\ 3k & \text{if } S \leq 3k; \end{cases}$

4. for each $\delta \in \Delta$, $|W_\delta| \leq \begin{cases} \frac{1}{k} S + 1 & \text{if } S > 3k, \\ 3k & \text{if } S \leq 3k. \end{cases}$

Now, we define the formula that results from applying *i iterations of the basic transformation* on a formula F of size S in the following way: Informally, first apply the basic transformation to F once, and then, in the formula that results, for each subformula in the outer portion apply the basic transformation again yielding a "new" inner portion and a "new" outer portion corresponding to that outer formula. Now, graft all the new inner portions onto the original inner portion and create a new outer portion consisting of all of the new outer portions. This results in a new formula (presented as an inner portion and an outer portion) on which the above steps can be applied again until i applications have been made.

More formally, for the formula F, define the sequence

$$\langle \Delta_i, E_i(w_\delta)_{\delta\in\Delta_i}, \{W_\delta : \delta \in \Delta_i\}\rangle \quad \text{for } i \geq 0$$

as follows: First, define $\langle \Delta_0, E_0(w_\delta)_{\delta\in\Delta_0}, \{W_\delta : \delta \in \Delta_0\}\rangle = \langle \{\varepsilon\}, w_\varepsilon, \{F\}\rangle$. Then, assuming that the tuple $\langle \Delta_i, E_i(w_\delta)_{\delta\in\Delta_i}, \{W_\delta : \delta \in \Delta_i\}\rangle$ has been defined for each $\delta \in \Delta_i$, apply the basic transformation to W_δ yielding

$$\langle \Lambda_\delta, D_\delta(w_\lambda)_{\lambda\in\Lambda_\delta}, \{W_\lambda : \lambda \in \Lambda_\delta\}\rangle.$$

Without loss of generality, assume that the sets Λ_δ are disjoint for distinct $\delta \in \Delta$. Then define

$$\langle \Delta_{i+1}, E_{i+1}(w_\delta)_{\delta\in\Delta_{i+1}}, \{W_\delta : \delta \in \Delta_{i+1}\}\rangle$$

as

1. $\Delta_{i+1} = \bigcup_{\delta\in\Delta_i} \Lambda_\delta$;
2. $E_{i+1}(w_\delta)_{\delta\in\Delta_{i+1}} = E_i(D_\delta(w_\lambda)_{\lambda\in\Lambda_\delta})_{\delta\in\Delta_i}$;
3. $\{W_\delta : \delta \in \Delta_{i+1}\} = \{W_\lambda : \lambda \in \bigcup_{\delta\in\Delta_i} \Lambda_\delta\}$.

This completes the definition of i iterations of the basic transformation.

It is straightforward to verify from this that, after applying i iterations of the basic transformation on F for $i > 0$, the following hold:

1. the resulting formula $E_i(W_\delta)_{\delta\in\Delta_i}$ is equivalent to F;
2. the depth of the inner portion is at most $3(k-1)i$;
3. each of the outer formulas has size at most (and depth less than)

$$\max(3k, k^{-i}S + k^{1-i} + k^{2-i} + \cdots + k^{-1} + 1) \le \max(3k, k^{-i}S + 2);$$

4. the size of the restructured formula is at most $2^i S + 4(k-1)(2^i - 1)$.

In particular, after $\lfloor \log_k S \rfloor$ iterations,

1. the depth of the inner portion is at most $3(k-1)\lfloor \log_k S \rfloor \le (3(k-1)/\log k)\log S$;
2. each of the outer formulas has size at most (and depth strictly less than)

$$\max(3k, k^{-\lfloor \log_k S\rfloor} S + 2) \le \max(3k, k+2) \le 3k;$$

3. the size of the restructured formula is at most

$$2^{\lfloor \log_k S\rfloor} S + 4(k-1)(2^{\lfloor \log_k S\rfloor} - 1) \le S^{1+1/\log k} + 4(k-1)(S^{1/\log k} - 1).$$

The theorem now follows from the above by noting that the depth of a restructured formula is at most the sum of the depth of its inner portion and the maximum depth of any of its outer portions. □

COROLLARY 5.3. *Over any field $\mathcal{F}$, for any fixed $\epsilon > 0$, for any division-free formula of size S with operations from $\{+, -, \times\} \cup \mathcal{F}$, there are equivalent formulas with*

(i) *depth $O(\log S)$ and size $O(S^{1+\epsilon})$,*
(ii) *depth $O(\log^{1+\epsilon} S)$ and size $S^{1+O(1/\log\log S)}$,*
(iii) *depth $O(S^\epsilon)$ and size $O(S)$.*

Proof. Apply Theorem 5.2, setting $k = 2^{1/\epsilon}$ (in the first case), $k = \log^\epsilon S$ (in the second case), and $k = S^{\epsilon/2}$ (in the third case). In all three cases the above depth and size bounds are direct consequences of the bounds given in the theorem. □

6. General algebraic formulas. We now consider algebraic formulas with divisions. The restructuring of partitioned formulas is similar to, but more complicated than, the method given above for the division-free case.

In the division-free case, Lemma 5.1 asserts that if $G(y)$ is read-once with respect to y then it can be expressed as $G(y) \equiv (P \times y) + Q$, where $|P|$ and $|Q|$ are approximately $|G(y)|$. In the more general case, where divisions may occur, it is easy to establish that $G(y)$ may either be expressed in the above form or in the form

$$G(y) \equiv \frac{(P \times y) + Q}{y + R}.$$

The difficulty arises in bounding the sizes $|P|$, $|Q|$, and $|R|$. The technique used in Lemma 5.1, which is inductive on the depth of y, does not readily generalize here because the quantities

$|P|$, $|Q|$, and $|R|$ may grow very quickly at each inductive step. For example, this may occur if $G(y) \equiv G_1(y) + G_2$, where

$$G_1(y) \equiv \frac{(P_1 \times y) + Q_1}{y + R_1}.$$

In this case, the inductive step would yield

$$G(y) \equiv \frac{((P_1 + G_2) \times y) + (Q_1 + (G_2 \times R_1))}{y + R_1},$$

whose size may be approximately double that of the formula for $G_1(y)$—even when $|G_2| = 1$.

Lemma 6.3 is an analogue of Lemma 5.1 and will be used in place of Lemma 5.1 to prove the main theorem. Lemmas 6.1 and 6.2 are used to prove Lemma 6.3; Lemma 6.1 also resembles Lemma 5.1, but is proved using a different technique that involves solving a system of linear equations for the aforementioned P, Q, and R. This system of equations includes three new auxiliary variables, z_1, z_2, and z_3—and the formulas obtained by this lemma are actually extended formulas in these new auxiliary variables. Since arbitrary substitution of constants for these variables can introduce divisions by zero, some care must be taken in eliminating them. Lemma 6.2 gives upper bounds for the number of constants whose substitution for z_1, z_2, or z_3 can cause problems. In Lemma 6.3 these bounds are applied, together with the results of Lemma 6.1, to obtain small formulas P, Q, and R that do *not* have any new auxiliary variables as subformulas, and can be used to replace $G(y)$ as described above.

LEMMA 6.1. *For any extended formula $G(y)$ that is read-once with respect to y, there exists an extended formula $H(y, g_1, g_2, g_3, z_1, z_2, z_3)$ and extended formulas $G(z_1)$, $G(z_2)$, $G(z_3)$ such that*

(i) $G(y) \equiv H(y, G(z_1), G(z_2), G(z_3), z_1, z_2, z_3)$;

(ii) $|G(z_1)|_{+\{z_1\}}$, $|G(z_2)|_{+\{z_2\}}$, $|G(z_3)|_{+\{z_3\}} \leq |G(y)| + 1$ (*i.e., the sizes of* $G(z_1)$, $G(z_2)$, *and* $G(z_3)$, *counting all variables, are bounded above by* $|G(y)| + 1$);

(iii) $|H(y, g_1, g_2, g_3, z_1, z_2, z_3)|_{\{y\}} = 1$ (*i.e.,* $H(y, g_1, g_2, g_3, z_1, z_2, z_3)$ *is read-once with respect to* y);

(iv) $|H(y, g_1, g_2, g_3, z_1, z_2, z_3)|_{\{g_1, g_2, g_3\}} \leq 44$;

(v) $|H(y, g_1, g_2, g_3, z_1, z_2, z_3)|_{+\{z_1, z_2, z_3\}} \leq 42$;

(vi) $\text{depth}_{\{y\}}(H(y, g_1, g_2, g_3, z_1, z_2, z_3)) \leq 5$;

(vii) $\text{depth}(H(y, g_1, g_2, g_3, z_1, z_2, z_3)) \leq 9$.

Proof. Part (ii) follows immediately from the fact that $G(y)$ is read-once with respect to y. Parts (i) and (iii)–(vii) are less trivial, and are proved below.

Since $G(y)$ is read-once with respect to y, there exist formulas P, Q, and R such that either

$$G(y) \equiv \frac{(P \times y) + Q}{y + R}$$

or

$$G(y) \equiv (P \times y) + Q.$$

The existence of P, Q, and R can be shown by considering the functions computed along the path in $G(y)$ from y to its root: each such function is the quotient of two affine linear functions of y. Although this establishes the existence of P, Q, and R, this does not lead to an efficient way of constructing these formulas; in the general case (with divisions), the resulting formula size may be exponential in $|G(y)|$. Instead, we use the method below.

First, we consider the case where

$$G(y) \equiv \frac{(P \times y) + Q}{y + R}.$$

Substituting three distinct new auxiliary variables z_1, z_2, and z_3 for y in this equation for $G(y)$ gives

$$G(z_i) = \frac{(P \times z_i) + Q}{z_i + R}, \quad i = 1, 2, 3.$$

This results in the system of equations

$$\begin{bmatrix} 1 & z_1 & -G(z_1) \\ 1 & z_2 & -G(z_2) \\ 1 & z_3 & -G(z_3) \end{bmatrix} \times \begin{bmatrix} Q \\ P \\ R \end{bmatrix} \equiv \begin{bmatrix} G(z_1) \times z_1 \\ G(z_2) \times z_2 \\ G(z_3) \times z_3 \end{bmatrix}.$$

Since z_1, z_2, and z_3 are new auxiliary variables and $G(y)$ is not of the form $(P \times y) + Q$, the columns of the above matrix are linearly independent and, therefore, this system of equations is nonsingular. This system is equivalent to the block lower triangular system

$$\begin{bmatrix} 1 & z_1 & -G(z_1) \\ 0 & z_2 - z_1 & G(z_1) - G(z_2) \\ 0 & z_3 - z_1 & G(z_1) - G(z_3) \end{bmatrix} \times \begin{bmatrix} Q \\ P \\ R \end{bmatrix} \equiv \begin{bmatrix} G(z_1) \times z_1 \\ G(z_2) \times z_2 - G(z_1) \times z_1 \\ G(z_3) \times z_3 - G(z_1) \times z_1 \end{bmatrix}.$$

To solve this, it is sufficient to find P and R such that

$$\begin{bmatrix} z_2 - z_1 & G(z_1) - G(z_2) \\ z_3 - z_1 & G(z_1) - G(z_3) \end{bmatrix} \times \begin{bmatrix} P \\ R \end{bmatrix} \equiv \begin{bmatrix} G(z_2) \times z_2 - G(z_1) \times z_1 \\ G(z_3) \times z_3 - G(z_1) \times z_1 \end{bmatrix},$$

and then use back substitution to find Q.

Let $(\vec{G}(\vec{z}), \vec{z})$ denote the six-tuple $(G(z_1), G(z_2), G(z_3), z_1, z_2, z_3)$ and $(\vec{g}, \vec{z})$ denote $(g_1, g_2, g_3, z_1, z_2, z_3)$. By Cramer's rule (applied to the above linear system)

$$\begin{aligned} P &\equiv \frac{\begin{vmatrix} G(z_2) \times z_2 - G(z_1) \times z_1 & G(z_1) - G(z_2) \\ G(z_3) \times z_3 - G(z_1) \times z_1 & G(z_1) - G(z_3) \end{vmatrix}}{\begin{vmatrix} z_2 - z_1 & G(z_1) - G(z_2) \\ z_3 - z_1 & G(z_1) - G(z_3) \end{vmatrix}} \\ &\equiv \frac{\hat{P}(\vec{G}(\vec{z}), \vec{z})}{D(\vec{G}(\vec{z}), \vec{z})}, \end{aligned}$$

where $\hat{P}(\vec{G}(\vec{z}), \vec{z})$ and $D(\vec{G}(\vec{z}), \vec{z})$ are equivalent to the numerator and denominator of the preceding expression, respectively. In particular, we can choose $\hat{P}(\vec{G}(\vec{z}), \vec{z})$ and $D(\vec{G}(\vec{z}), \vec{z})$ where $\hat{P}(\vec{g}, \vec{z})$ and $D(\vec{g}, \vec{z})$ are as shown in Figs. 3 and 4. Clearly, $\text{depth}(\hat{P}(\vec{g}, \vec{z})) = 4$, $\text{depth}(D(\vec{g}, \vec{z})) = 3$, $|\hat{P}(\vec{g}, \vec{z})| = 0$, $|\hat{P}(\vec{g}, \vec{z})|_{\{\vec{g}\}} = 8$, $|\hat{P}(\vec{g}, \vec{z})|_{\{\vec{z}\}} = 4$, $|D(\vec{g}, \vec{z})| = 0$, $|D(\vec{g}, \vec{z})|_{\{\vec{g}\}} = 4$, and $|D(\vec{g}, \vec{z})|_{\{\vec{z}\}} = 4$.

Similarly,

$$\begin{aligned} R &\equiv \frac{\begin{vmatrix} z_2 - z_1 & G(z_2) \times z_2 - G(z_1) \times z_1 \\ z_3 - z_1 & G(z_3) \times z_3 - G(z_1) \times z_1 \end{vmatrix}}{\begin{vmatrix} z_2 - z_1 & G(z_1) - G(z_2) \\ z_3 - z_1 & G(z_1) - G(z_3) \end{vmatrix}} \\ &\equiv \frac{\hat{R}(\vec{G}(\vec{z}), \vec{z})}{D(\vec{G}(\vec{z}), \vec{z})}, \end{aligned}$$

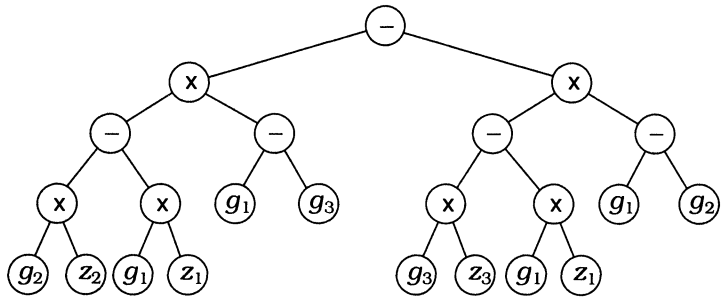


FIG. 3. *Formula* $\hat{P}(\vec{g}, \vec{z})$.

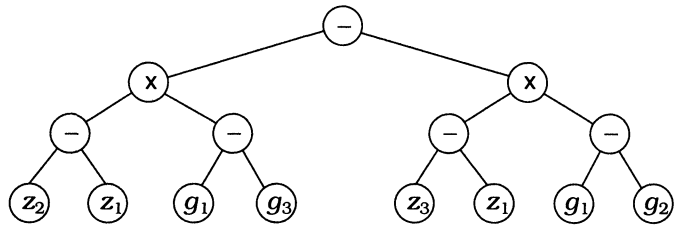


FIG. 4. *Formula* $D(\vec{g}, \vec{z})$.

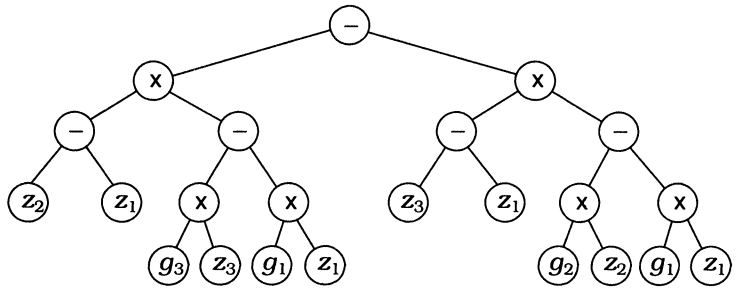


FIG. 5. *Formula* $\hat{R}(\vec{g}, \vec{z})$.

where formula $\hat{R}(\vec{g}, \vec{z})$ is as shown in Fig. 5. Clearly, $\text{depth}(\hat{R}(\vec{g}, \vec{z})) = 4$, $|\hat{R}(\vec{g}, \vec{z})| = 0$, $|\hat{R}(\vec{g}, \vec{z})|_{\{\vec{g}\}} = 4$, and $|\hat{R}(\vec{g}, \vec{z})|_{\{\vec{z}\}} = 8$.

It can be shown by back substitution that

$$Q(\vec{G}(\vec{z}), \vec{z}) \times D(\vec{G}(\vec{z}), \vec{z}) \equiv z_1 \times G(z_1) \times D(\vec{G}(\vec{z}), \vec{z}) - z_1 \times \hat{P}(\vec{G}(\vec{z}), \vec{z}) + G(z_1) \times \hat{R}(\vec{G}(\vec{z}), \vec{z}).$$

It is straightforward (but tedious) to confirm that this is equivalent to $\hat{Q}(\vec{G}(\vec{z}), \vec{z})$ for the formula $\hat{Q}(\vec{g}, \vec{z})$ as shown in Fig. 6. It is easily checked that $\text{depth}(\hat{Q}(\vec{g}, \vec{z})) = 4$, $|\hat{Q}(\vec{g}, \vec{z})| = 0$, $|\hat{Q}(\vec{g}, \vec{z})|_{\{\vec{g}\}} = 8$, and $|\hat{Q}(\vec{g}, \vec{z})|_{\{\vec{z}\}} = 6$.

Clearly, the desired $H(y, \vec{g}, \vec{z})$ can be expressed in terms of $\hat{P}(\vec{g}, \vec{z})$, $\hat{Q}(\vec{g}, \vec{z})$, $\hat{R}(\vec{g}, \vec{z})$, and $D(\vec{g}, \vec{z})$; however, prior to completing the construction, we restructure the expression $((P \times y) + Q) \div (y + R)$ so that it is read-once with respect to y. This is accomplished by performing a "polynomial division" of $y + R$ into $(P \times y) + Q$, resulting in the identity $((P \times y) + Q) \div (y + R) = \tilde{H}(y, P, Q, R, D)$ for the formula $\tilde{H}(y, p, q, r, d)$ given in Fig. 7.

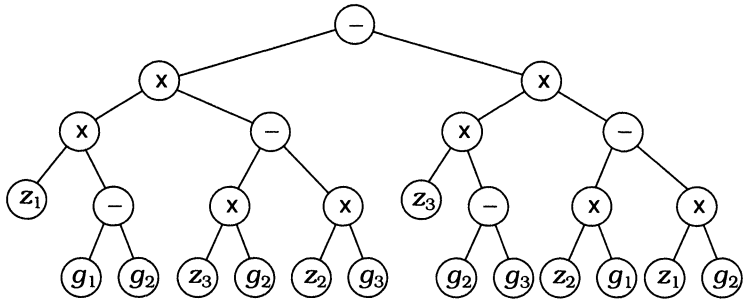


FIG. 6. *Formula* $\hat{Q}(\vec{g}, \vec{z})$.

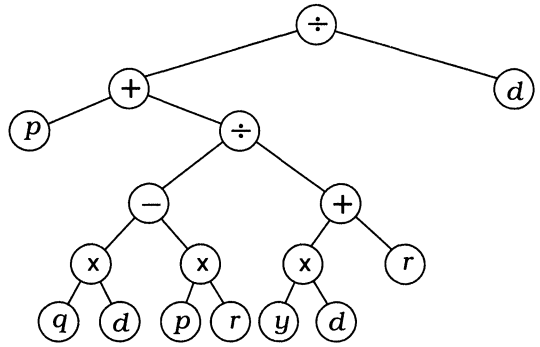


FIG. 7. *Formula* $\tilde{H}(y, p, q, r, d)$.

Set $H(y, \vec{g}, \vec{z}) = \tilde{H}(y, \hat{P}(\vec{g}, \vec{z}), \hat{Q}(\vec{g}, \vec{z}), \hat{R}(\vec{g}, \vec{z}), D(\vec{g}, \vec{z}))$; then it is easily checked that $\text{depth}(H(y, \vec{g}, \vec{z})) = 9$, $|H(y, \vec{g}, \vec{z})| = 0$, $|H(y, \vec{g}, \vec{z})|_{\{y\}} = 1$, $|H(y, \vec{g}, \vec{z})|_{\{\vec{g}\}} = 44$, and $|H(y, \vec{g}, \vec{z})|_{\{\vec{z}\}} = 42$ as claimed. Furthermore, it follows by the construction of these formulas that

$$G(y) \equiv H(y, \vec{G}(\vec{z}), \vec{z}),$$

which completes the proof for the case where $G(y) \equiv ((P \times y) + Q) \div (y + R)$.

The proof of the simpler case where $G(y) \equiv (P \times y) + Q$ is straightforward and omitted here. □

The formulas introduced in Lemma 6.1 introduce new variables, z_1, z_2, and z_3. The next lemmas establish that these can be replaced by elements of the ground field $\mathcal{F}$, provided $\mathcal{F}$ is sufficiently large.

LEMMA 6.2. (i) *Suppose* $G(z) = G_L(z) * G_R(z)$ *for* $* \in \{+, -, \times, \div\}$ *and no proper subformula of* $G(z)$ *is identically zero. Let* d_L *(respectively,* d_R*) be an upper bound on the degree in* z *of the numerator and denominator of the rational function* $G_L(z)$ *(respectively,* $G_R(z)$*). If* $* \in \{+, -\}$ *then there are at most* $d_L + d_R$ *elements* $\zeta \in \mathcal{F}$ *such that* $G(\zeta)$ *is identically zero but none of* $G_L(\zeta)$, $G_R(\zeta)$, *or any of their subformulas are identically zero. If* $* \in \{\times, \div\}$ *then there are no elements* $\zeta \in \mathcal{F}$ *such that* $G(\zeta)$ *is identically zero but none of* $G_L(\zeta)$, $G_R(\zeta)$, *or any of their proper subformulas are identically zero.*

(ii) *Suppose* $G(z)$ *is read-once with respect to* z *and that no subformula of* $G(z)$ *is identically zero. Then there are at most* $1 + \text{depth}(G(z)) \leq |G(z)|$ *elements* $\zeta \in \mathcal{F}$ *such that a subformula of* $G(\zeta)$ *is identically zero.*

Proof. The first claim is easily verified by expressing the numerator of $G(z)$ as a function of the numerators and denominators of $G_L(z)$ and $G_R(z)$. If $* \in \{+, -\}$ then this numerator

is a nonzero polynomial with degree at most $d_L + d_R$ in z. If $* \in \{\times, \div\}$ then the numerator is a product of numerators or of a numerator and denominator of subformulas of $G_L(z)$ and $G_R(z)$.

The second claim can be proved using induction on the depth of $G(z)$ and the fact that, if $G(z)$ is read-once with respect to z, then every subformula of $G(z)$ that includes z is the quotient of two affine linear functions of z, while every other subformula of $G(z)$ has degree zero in z. □

LEMMA 6.3. *Suppose the extended formula $G(y)$ is read-once with respect to y, $|\mathcal{F}| \geq |G(y)| + 6$, and let S be a subset of $\mathcal{F}$ with at least $|G(y)| + 6$ distinct elements. Then there exist an extended formula $\hat{H}(y, a, b, c)$ and formulas A, B, and C such that*

(i) $G(y) \equiv \hat{H}(y, A, B, C)$;
(ii) $|A|, |B|, |C| \leq |G(y)| + 1$;
(iii) $|(\hat{H}(y, a, b, c)|_{\{y\}} = 1$ *(i.e., $\hat{H}(y, a, b, c)$ is read-once with respect to y)*;
(iv) $|\hat{H}(y, a, b, c)|_{\{a,b,c\}} \leq 44$;
(v) $|\hat{H}(y, a, b, c)| \leq 42$;
(vi) $\text{depth}_{\{y\}}(\hat{H}(y, a, b, c)) \leq 5$;
(vii) $\text{depth}(\hat{H}(y, a, b, c)) \leq 9$;
(viii) *the only constants occurring as a subformula of $\hat{H}(y, a, b, c)$, A, B, or C either occur as a subformula of $G(y)$ or belong to S.*

Proof. As argued in the proof of Lemma 6.1, since $G(y)$ is read-once with respect to y, there exist formulas P, Q, and R such that either

$$G(y) \equiv \frac{(P \times y) + Q}{y + R}$$

or

$$G(y) \equiv (P \times y) + Q.$$

Suppose the first case holds and let $H(y, g_1, g_2, g_3, z_1, z_2, z_3)$ be the formula that exists by applying Lemma 6.1 to $G(y)$. If, as in the proof of Lemma 6.1,

$$D(g_1, g_2, g_3, z_1, z_2, z_3) = \begin{vmatrix} z_2 - z_1 & g_1 - g_2 \\ z_3 - z_1 & g_1 - g_3 \end{vmatrix} = (z_2 - z_1) \times (g_1 - g_3) - (z_3 - z_1) \times (g_1 - g_2),$$

then for any $\zeta_1, \zeta_2, \zeta_3 \in \mathcal{F}$, the formula $H(y, G(\zeta_1), G(\zeta_2), G(\zeta_3), \zeta_1, \zeta_2, \zeta_3)$ is well defined and equivalent to $G(y)$ provided that $D(G(\zeta_1), G(\zeta_2), G(\zeta_3), \zeta_1, \zeta_2, \zeta_3)$ is well defined and not equivalent to the zero function (again, see the proof of Lemma 6.1 for details).

We next show that $\zeta_1, \zeta_2, \zeta_3 \in S$ exist with the above properties. To begin, we count the number of elements $\psi \in \mathcal{F}$ such that either $D(G(\psi), G(z_2), G(z_3), \psi, z_2, z_3)$ or one of its subformulas is identically zero. By Lemma 6.2 (ii) there are at most $|G(y)|$ values $\psi \in \mathcal{F}$ such that $G(\psi)$ or one of its subformulas is identically zero. Since $D(\vec{G}(\vec{z}), \vec{z})$ is not identically zero, the value of $G(y)$ depends on the value of y. Therefore, there are no additional elements $\psi \in \mathcal{F}$ such that either $G(\psi) - G(z_2)$ or $G(\psi) - G(z_3)$ or any of their subformulas is identically zero. There is only one element $\psi \in \mathcal{F}$ for which $z_2 - \psi$, $z_3 - \psi$, or any of their subformulas is identically zero (namely, $\psi = 0$). By Lemma 6.2 (i) there are no additional values ψ for which either $(z_2 - \psi) \times (G(\psi) - G(z_3))$ or $(z_3 - \psi) \times (G(\psi) - G(z_2))$ is identically zero and, since both of these are equivalent to rational functions whose numerators and denominators have degree at most two in ψ, there are at most four additional values ψ for which $D(G(\psi), G(z_2), G(z_3), \psi, z_2, z_3)$ is identically zero. Thus there are at most $|G(y)| + 5$ values $\psi \in \mathcal{F}$ such that either $D(G(\psi), G(z_2), G(z_3), \psi, z_2, z_3)$ or one of its subformulas is

identically zero. Since $|\mathcal{S}| \geq |G(y)| + 6$, it is possible to choose an element $\zeta_1 \in \mathcal{S}$ such that neither $D(G(\zeta_1), G(z_2), G(z_3), \zeta_1, z_2, z_3)$ nor any of its subformulas is identically zero.

We next count the number of elements ψ of the field $\mathcal{F}$ such that either the formula $D(G(\zeta_1), G(\psi), G(z_3), \zeta_1, \psi, z_3)$ or one of its subformulas is identically zero. By Lemma 6.2 there are at most $|G(y)| + 1$ values ψ such that either $G(\zeta_1) - G(\psi)$ or one of its subformulas is identically zero, and at most two values ψ such that either $\psi - \zeta_1$ or one of its subformulas is identically zero. Formulas $G(\zeta_1) - G(z_3)$ and $z_3 - \zeta_1$ are independent of ψ. Finally, since both $(\psi - \zeta_1) \times (G(\zeta_1) - G(z_3))$ and $(z_3 - \zeta_1) \times (G(\zeta_1) - G(\psi))$ are equivalent to rational functions whose numerator and denominator are affine linear in ψ, there are at most two additional values ψ such that $D(G(\zeta_1), G(\psi), G(z_3), \zeta_1, \psi, z_3)$ is identically zero. Again, since $|\mathcal{S}| \geq |G(y)| + 6$, it is possible to choose $\zeta_2 \in \mathcal{S}$ such that neither $D(G(\zeta_1), G(\zeta_2), G(z_3), \zeta_1, \zeta_2, z_3)$ nor any of its subformulas is identically zero. It follows by virtually the same argument that it is also possible to choose $\zeta_3 \in \mathcal{S}$ such that neither $D(G(\zeta_1), G(\zeta_2), G(\zeta_3), \zeta_1, \zeta_2, \zeta_3)$ nor any of its subformulas is identically zero as well.

Now, set

$$\hat{H}(y, a, b, c) = H(y, a, b, c, \zeta_1, \zeta_2, \zeta_3),$$

$$A = G(\zeta_1), \quad B = G(\zeta_2), \quad C = G(\zeta_3).$$

Clearly, $\hat{H}(y, A, B, C) \equiv H(y, G(\zeta_1), G(\zeta_2), G(\zeta_3), \zeta_1, \zeta_2, \zeta_3) \equiv G(y)$, the size bounds stated above for A, B, and C follow immediately from the definitions of these formulas, and the size and depth bounds for $\hat{H}(a, b, c)$ and $\hat{H}(A, B, C)$ follow directly from those given for $H(y, g_1, g_2, g_3, y_1, y_2, y_3)$ in the statement of Lemma 6.1.

The simpler case where $G(y) \equiv (P \times y) + Q$ is handled similarly. □

THEOREM 6.4. *For any formula F of size $S \leq |\mathcal{F}| - 6$, any subset $\mathcal{S}$ of $\mathcal{F}$ with size at least $S + 6$, and any integer $k \geq 2$, there exists a formula G that is equivalent to F and has depth bounded by*

$$\left(\frac{3(2k-1)}{\log k}\right) \log S + 6k - 1$$

and size bounded by

$$S^{1+\log 44/\log k} + 4(k-1)(S^{\log 44/\log k} - 1)$$

such that the only constants appearing as a subformula of G either appear as a subformula of F or belong to $\mathcal{S}$.

For any formula F of size $S > |\mathcal{F}| - 6$ and any integer $k \geq 2$, there exists a formula G that is equivalent to F and has depth bounded by

$$\left(\frac{3(2k-1)}{\log k} + 2\right) \log S + 6k + 3$$

and size bounded by

$$(S^{1+\log 44/\log k} + 4(k-1)(S^{\log 44/\log k} - 1)) \left(\frac{2 \log S}{\log |\mathcal{F}|} + 5\right).$$

Proof. Let F be an arbitrary formula of size $S \leq |\mathcal{F}| - 6$, $\mathcal{S}$ be a subset of $\mathcal{F}$ with at least $S + 6$ elements, and k be an arbitrary integer such that $k \geq 2$. The construction of a small

size, small depth formula equivalent to F is similar in this case to the construction given in the proof of Theorem 5.2.

As before, we first define a basic transformation of F that yields a restructured formula equivalent to F—again presented as an index set Δ, an inner portion $E(w_\delta)_{\delta\in\Delta}$ which is an extended formula over a set of auxiliary variables w_δ (for $\delta \in \Delta$), and an outer portion including a formula W_δ for each auxiliary variable w_δ of the inner portion, such that $F \equiv E(W_\delta)_{\delta\in\Delta}$.

If $S \leq 6k$ then it is sufficient to set $\Delta = \{\varepsilon\}$, $E(W_\varepsilon) = w_\varepsilon$, and $W_\varepsilon = F$. Otherwise, we apply Lemma 4.1 to F and k, yielding sets *Interior*, *Boundary*, extended formulas $G_\alpha(y)$ and operations $*_\alpha$ for each $\alpha \in$ *Interior*, and formulas G_β for each $\beta \in$ *Boundary*, satisfying all the conditions of the lemma. Second, we apply Lemma 6.3 (instead of Lemma 5.1) to each $G_\alpha(y)$, yielding $\hat{H}_\alpha(y, a, b, c)$, A_α, B_α, and C_α, satisfying the conditions of this lemma.

The inductive definition of the inner portion $E(w_\delta)_{\delta\in\Delta}$ is almost identical to the one given in the proof of Theorem 5.2: for each $\gamma \in$ *Interior* $\cup$ *Boundary*, define an extended formula V_γ (with auxiliary variables a_α, b_α, and c_α for $\alpha \in$ *Interior* and g_β for $\beta \in$ *Boundary*) by

$$V_\gamma = \begin{cases} \hat{H}_\gamma(V_{\gamma 0} *_\gamma V_{\gamma 1}, a_\gamma, b_\gamma, c_\gamma) & \text{if } \gamma \in \textit{Interior}, \\ g_\gamma & \text{if } \gamma \in \textit{Boundary}. \end{cases}$$

Then set the outer portion as V_ε.

For each $\alpha \in$ *Interior*, set the outer formulas corresponding to the auxiliary variables a_α, b_α, and c_α as A_α, B_α, and C_α, respectively, and for each $\beta \in$ *Boundary*, set the outer formula corresponding to g_β as G_β.

As before, to simplify notation at later stages, rename the auxiliary variables of the inner portion as w_δ, where δ ranges over some index set Δ (so $V_\varepsilon = E(w_\delta)_{\delta\in\Delta}$) and rename the corresponding outer formulas as W_δ to complete the definition of the basic transformation of F.

It is again clear that $F \equiv E(W_\delta)_{\delta\in\Delta}$, since $\hat{H}_\alpha(y, A_\alpha, B_\alpha, C_\alpha) \equiv G_\alpha(y)$ for all $\alpha \in$ *Interior*.

Since $\text{depth}_{\{y\}}(\hat{H}_\alpha(y, a_\alpha, b_\alpha, c_\alpha)) \leq 5$ and $\text{depth}(\hat{H}_\alpha(y, a_\alpha, b_\alpha, c_\alpha)) \leq 9$ for each $\alpha \in$ *Interior*, and since *Interior* $\subseteq \{0, 1\}^{\leq k-2}$, it is straightforward to verify that

$$\text{depth}(E(w_\delta)_{\delta\in\Delta}) \leq 6(k-2) + 9 = 3(2k-1).$$

Now consider the size of $E(W_\delta)_{\delta\in\Delta}$. Since the size of $\hat{H}_\alpha(a_\alpha, b_\alpha, c_\alpha)$ is at most 42, $|\hat{H}_\alpha(y, a_\alpha, b_\alpha, c_\alpha)|_{\{a_\alpha, b_\alpha, c_\alpha\}} \leq 44$, and $|A_\alpha|, |B_\alpha|, |C_\alpha| \leq |G_\alpha(y)| + 1$ for all $\alpha \in$ *Interior*,

$$|\hat{H}_\alpha(y, A_\alpha, B_\alpha, C_\alpha)| \leq 42 + 44(|G_\alpha(y)| + 1) = 44|G_\alpha(y)| + 86.$$

It follows by the definition of $E(W_\delta)_{\delta\in\Delta}$ that

$$\begin{aligned} |E(W_\delta)_{\delta\in\Delta}| &= \sum_{\alpha\in\textit{Interior}} |\hat{H}_\alpha(y, A_\alpha, B_\alpha, C_\alpha)| + \sum_{\beta\in\textit{Boundary}} |G_\beta| \\ &\leq 44 \sum_{\alpha\in\textit{Interior}} |G_\alpha(y)| + 86|\textit{Interior}| + \sum_{\beta\in\textit{Boundary}} |G_\beta| \\ &\leq 44S + 172(k-1), \end{aligned}$$

since $S = \sum_{\alpha\in\textit{Interior}} |G_\alpha(y)| + \sum_{\beta\in\textit{Boundary}} |G_\beta|$ and $|\textit{Interior}| \leq 2k - 2$.

Finally, $|W_\delta| \leq \lfloor \frac{1}{k}|F| + 1 \rfloor \leq \frac{1}{k}S + 1$ for all $\delta \in \Delta$ as before.

In summary, the basic transformation converts a formula F of size S to one denoted by the three-tuple $\langle \Delta, E(w_\delta)_{\delta\in\Delta}, \{W_\delta : \delta \in \Delta\}\rangle$, where

1. $E(W_\delta)_{\delta\in\Delta} \equiv F$;

2. $\text{depth}(E(w_\delta)_{\delta\in\Delta}) \le \begin{cases} 3(2k-1) & \text{if } S > 6k, \\ 0 & \text{if } S \le 6k; \end{cases}$
3. $|E(W_\delta)_{\delta\in\Delta}| \le \begin{cases} 44S + 172(k-1) & \text{if } S > 6k, \\ 6k & \text{if } S \le 6k; \end{cases}$
4. For each $\delta \in \Delta$, $|W_\delta| \le \begin{cases} \frac{1}{k}S + 1 & \text{if } S > 6k, \\ 6k & \text{if } S \le 6k. \end{cases}$

We next define a sequence

$$\langle \Delta_i, E_i(w_\delta)_{\delta\in\Delta_i}, \{W_\delta : \delta \in \Delta_i\}\rangle \quad \text{for } i \ge 0$$

of formulas equivalent to F corresponding to iterations of the basic transformation on F—in precisely the same way as this is defined in the proof of Theorem 5.2. It is straightforward to verify that, after applying i iterations of the basic transformation on F, for $i > 0$, the following hold:

1. the resulting formula $E_i(W_\delta)_{\delta\in\Delta_i}$ is equivalent to F;
2. the depth of the inner portion $E_i(w_\delta)_{\delta\in\Delta_i}$ is at most $3(2k-1)i$;
3. each of the outer formulas has size at most (and depth less than)

$$\max(6k, k^{-i}S + k^{1-i} + k^{2-i} + \cdots + k^{-1} + 1) \le \max(6k, k^{-i}S + 2);$$

4. the size of the restructured formula $E_i(W_\delta)_{\delta\in\Delta_i}$ is at most

$$44^i S + \left(\frac{44^i - 1}{44 - 1}\right) 172(k-1)(44^i - 1) \le 44^i S + 4(k-1)(44^i - 1).$$

In particular, after $\lfloor \log_k S \rfloor = \lfloor \log S / \log k \rfloor$ iterations,

1. each of the outer formulas has size at most (and depth less than)

$$\max(6k, k^{-\lfloor \log_k S\rfloor} S + 2) \le \max(6k, k + 2) \le 6k;$$

2. the size of the restructured formula is at most

$$\begin{aligned} &44^{\lfloor \log_k S\rfloor} S + 4(k-1)(44^{\lfloor \log_k S\rfloor} - 1) \\ &\quad \le S^{1+\log 44/\log k} + 4(k-1)(S^{\log 44/\log k} - 1). \end{aligned}$$

The results of the theorem for the case $S \le |\mathcal{F}| - 6$ now follow from the above by noting that the depth of a restructured formula is at most the sum of the depth of its inner portion and the maximum depth of any of its outer formulas.

Now suppose that $\mathcal{F}$ is finite and that F is a formula with size $S > \max(|\mathcal{F}| - 6, 1)$. Let $\mathcal{E} = \mathcal{F}(x_1)$ and consider F as a formula of size S over the infinite field $\mathcal{E}$; the only "constants" arising as subformulas of F are x_1 and elements of the small field $\mathcal{F}$. Let $\mathcal{S} \subset \mathcal{E}$ include all elements of $\mathcal{F}[x_1]$ whose degree in x_1 is at most $\log_{|\mathcal{F}|}(S+6)$; clearly, $|\mathcal{S}| \ge S + 6$, and (by the claim for formulas over large fields) there exists a formula $\hat{G}$ equivalent to F that has depth bounded by

$$\frac{3(2k-1)}{\log k} \log S + 6k - 1$$

and size bounded by

$$S^{1+\log 44/\log k} + 4(k-1)(S^{\log 44/\log k} - 1),$$

such that the only constants appearing as a subformula of $\hat{G}$ either appear as a subformula of F (hence are x_1 or belong to $\mathcal{F}$) or belong to $\mathcal{S}$. Now, each element of $\mathcal{S}$ is equivalent to a formula

(with constants in $\mathcal{F}$ and variable x_1) with size at most $2\log_{|\mathcal{F}|}(S+6)+1 \le 2\log_{|\mathcal{F}|} S+5$ and depth at most $2\log_{|\mathcal{F}|}(S+6) \le 2\log_{|\mathcal{F}|} S+4$. Therefore, the formula $\hat{G}$ can be used to obtain an equivalent formula G with constants in $\mathcal{F}$ and variables $x_1, x_2, \ldots$ such that $|G| \le |\hat{G}|(2\log_{|\mathcal{F}|} S+5)$ and $\text{depth}(G) \le \text{depth}(\hat{G}) + 2\log_{|\mathcal{F}|} S+4$ as required to prove the claim for the case in which $\mathcal{F}$ is small. □

COROLLARY 6.5. *Over any field $\mathcal{F}$, for any fixed $\epsilon > 0$, for any formula of size S with operations from $\{+, -, \times, \div\} \cup \mathcal{F}$, there are equivalent formulas with*

(i) *depth $O(\log S)$ and size $O(S^{1+\epsilon})$,*

(ii) *depth $O(\log^{1+\epsilon} S)$ and size $S^{1+O(1/\log\log S)}$,*

(iii) *depth $O(S^{\epsilon})$ and size* $\begin{cases} O(S) & \text{if } |\mathcal{F}| \ge S, \\ O(S\frac{\log S}{\log |\mathcal{F}|}) & \text{if } |\mathcal{F}| < S. \end{cases}$

Proof. Apply Theorem 6.4 setting $k = 45^{1/\epsilon}$ (in the first case), $k = \log^{\epsilon} S$ (in the second case), and $k = S^{\epsilon/2}$ (in the third case). □

7. Simple formulas. Kosaraju [7] showed that any simple formula F is equivalent to a division-free formula $\hat{F}$ of depth at most $\log|F| + 2\sqrt{\log|F|} + d$ for some constant d; clearly, such a formula $\hat{F}$ can have size at most $c|F|^{1+2/\sqrt{\log|F|}} \in O(|F|^{1+\epsilon})$ for some constant c and arbitrary $\epsilon > 0$. As shown below, Brent's construction and the techniques used to prove Theorems 5.2 and 6.4 can be used to improve the bound on formula size implied by Kosaraju [7] for restructuring simple formulas.

LEMMA 7.1. *Suppose $G(y)$ is read-once with respect to y. If $|G(y)| = 0$ then $G(y) = y$. If $|G(y)| \ge 1$ then there exist division-free formulas A and B such that $|A|, |B| \le |G(y)|$, and either*

$$G(y) \equiv A \times y + B \qquad \text{or} \qquad G(y) \equiv A \times y - B.$$

Furthermore, if $G(y)$ is simple then B is simple as well, the only operation in A is $\times$, and the depth of A is at most $\lfloor \log |G(y)| \rfloor$.

Proof. Clearly, $G(y) = y$ if $|G(y)| = 0$ and $G(y)$ is read-once with respect to y.

Now suppose that $|G(y)| \ge 1$. We first show that formulas A and B exist with all the stated properties, except that the depth of A may be larger than claimed when $G(y)$ is simple. This is proved using induction on the depth d of y in $G(y)$. Since $|G(y)| \ge 1$, $d \ge 1$.

If $d = 1$ then either

1. $G(y) = y + G'$ (or $G' + y$),
2. $G(y) = y - G'$,
3. $G(y) = G' - y$, or
4. $G(y) = y \times G'$ (or $G' \times y$)

for some formula G' such that $|G'| = |G(y)|$. Clearly, when $G(y)$ is simple, G' is either an input or a constant in case 4. In the first of these cases $G(y) = A \times y + B$ for $A = 1$ and $B = G'$. In the second case $G(y) = A \times y - B$ for the same choice of A and B. In the third case $G(y) = A \times y + B$ for $A = -1$ and $B = G'$. Finally, in the fourth case $G(y) = A \times y + B$ for $A = G'$ and $B = 0$. In each case, it is clear that $|A|, |B| \le |G(y)|$ and, if $G(y)$ is simple, then so is B, and the only operation in A is $\times$ as desired.

Now suppose that $d > 1$; then there exist formulas $G_1(y)$ and G_2 (both simple if $G(y)$ is simple) such that $G_1(y)$ is read-once with respect to y, the depth of y in $G_1(y)$ is $d - 1 > 0$, $|G_1(y)| \ge 1$, $|G_2| \ge 1$, and $|G_1(y)| + |G_2| = |G(y)|$, so $1 \le |G_1(y)|, |G_2| \le |G(y)| - 1$. Furthermore, either

1. $G(y) = G_1(y) + G_2$ (or $G_2 + G_1(y)$),
2. $G(y) = G_1(y) - G_2$,
3. $G(y) = G_2 - G_1(y)$, or

4. $G(y) = G_1(y) \times G_2$ (or $G_2 \times G_1(y)$).

If $G(y)$ is simple then, clearly, G_2 is either an input or a constant in case 4.

By the inductive hypothesis there exist division-free formulas A_1 and B_1 such that $|A_1|, |B_1| \leq |G_1(y)|$ and either $G_1(y) \equiv A_1 \times y + B_1$ or $G_1(y) \equiv A_1 \times y - B_1$, and such that B_1 is simple and the only operation in A_1 is $\times$ if $G_1(y)$ is simple. Now, since either $G_1(y) \equiv A_1 \times y + B_1$ or $G_1(y) \equiv A_1 \times y - B_1$, and $G(y)$ can assume any of the above four forms in each case, there are eight ways in which $G(y)$ can be constructed from A_1, B_1, y, and G_2. It is straightforward to check that in all eight cases, either $G(y) \equiv A \times y + B$ or $G(y) \equiv A \times y - B$, where A is one of A_1, $-1 \times A_1$, or $A_1 \times G_2$, and B is either $G_2 + B$, $G_2 - B$, or $G_2 \times B$. Clearly, $|A|, |B| \leq |G(y)|$ and A and B are division-free in each case. Furthermore, if $G_1(y)$ is simple then A is only set to be $A_1 \times G_2$ when G_2 is either an input or a constant.

Finally, suppose $G(y)$ is simple. Then the above formula A is simply a product of all of its inputs and constants. Since multiplication is associative, this can be replaced by an equivalent balanced formula of the same size. The depth of this new formula is at most the floor of the logarithm of its size as is needed to complete the proof. □

Recall that Lemma 5.1 established that if $G(y)$ is division-free and read-once with respect to y then $G(y) \equiv A \times y + B$ for division-free formulas A and B of size at most $|G(y)| + 1$. The results of Theorem 5.2 can be improved slightly if Lemma 7.1 is applied in the construction used to prove the theorem instead of Lemma 5.1. In particular, if this substitution is made then the resulting "basic transformation" converts a division-free formula F of size S into $\langle \Delta, E(w_\delta)_{\delta \in \Delta}, \{W_\delta : \delta \in \Delta\} \rangle$, such that $E(W_\delta)_{\delta \in \Delta} \equiv F$ and the depth of $E(w_\delta)_{\delta \in \Delta}$ and the size of each auxiliary formula W_δ are as described in the proof of the theorem, but where

$$|E(W_\delta)_{\delta \in \Delta}| \leq \begin{cases} 2S & \text{if } S > 3k, \\ 3k & \text{if } S \leq 3k. \end{cases}$$

Consequently, i iterations of the basic transformation produce a restructured formula of size at most $2^i S$, with the remaining properties described in the proof of Theorem 5.2—and the size bound of Theorem 5.2(iii) can be improved to "$|G| \leq S^{1+1/\log k}$". However, this change does not lead to an improvement in Corollary 5.3 (or to any other notable benefits).

THEOREM 7.2. *For any simple formula F there exists an equivalent division-free formula G (not generally simple) with depth at most* $3 \log |F|$ *such that* $|G| \leq \frac{1}{2}|F| \log |F| + |F|$.

Proof. We prove the result by induction on the size of F. The result is trivial if $|F| \leq 2$ since it is sufficient to set $G = F$.

Suppose $|F| > 2$ and set $m = \lceil \frac{1}{2}|F| \rceil$. By Lemma 3.1 there exists an extended formula $G(y)$ that is read-once with respect to y, formulas U and V, and an operation $*$ such that $F = G(U * V)$, $|G(y)| \leq |F| - m = \lfloor \frac{1}{2}|F| \rfloor$, and $|U|, |V| \leq m - 1 \leq \lfloor \frac{1}{2}|F| \rfloor$. Since F is simple, $G(y)$, U, and V are simple as well.

Suppose $G(y) \neq y$. Then, by Lemma 7.1, there exist division-free formulas A and B such that $|A|, |B| \leq |G(y)|$, B is simple, $\text{depth}(A) \leq \lfloor \log |G(y)| \rfloor \leq \lfloor \log |F| \rfloor - 1$, and either $G(y) \equiv (A \times y) + B$ or $G(y) \equiv (A \times y) - B$. Consequently, $F \equiv H(U, V, A, B)$, where either $H(u, v, a, b) = (a \times (u * v)) + b$ or $H(u, v, a, b) = (a \times (u * v)) - b$. By the inductive hypothesis, U is equivalent to a formula $\hat{U}$ such that the depth of $\hat{U}$ is at most $3 \log |U|$ and $|\hat{U}| \leq |U| + \frac{1}{2}|U| \log |U|$. Similarly, V is equivalent to a formula $\hat{V}$ whose depth is at most $3 \log |V|$ and whose size is at most $|V| + \frac{1}{2}|V| \log |V|$, and B is equivalent to a formula $\hat{B}$ with depth at most $3 \log |B|$ and size at most $|B| + \frac{1}{2}|B| \log |B|$. Set

$$G = H(\hat{U}, \hat{V}, A, \hat{B}).$$

Then $G \equiv F$, and the depth of G is the maximum of $1+\text{depth}(\hat{B}), 2+\text{depth}(A), 3+\text{depth}(\hat{U})$, and $3+\text{depth}(\hat{V})$. Since the depths of $\hat{U}$, $\hat{V}$, and $\hat{B}$ are at most $3\log\lfloor(|F|/2)\rfloor$ and the depth of A is at most $(\log|F|)-1 \le 3\log|F|-2$, the depth of G is at most $3\log|F|$. Also,

$$\begin{aligned} |G| &\le |A|+|\hat{U}|+|\hat{V}|+|\hat{B}| \\ &\le \lfloor\tfrac{1}{2}|F|\rfloor + (|U|+|V|+|B|) + \tfrac{1}{2}(|U|+|V|+|B|)\log\lfloor\tfrac{1}{2}|F|\rfloor \\ &\le \lfloor\tfrac{1}{2}|F|\rfloor + |F| + \tfrac{1}{2}|F|((\log|F|)-1) \\ &\le |F| + \tfrac{1}{2}|F|\log|F| \end{aligned}$$

as desired.

If $G(y) = y$ then $F = U * V$ and the result follows by a similar (but simpler) argument. □

It is possible to obtain an improved version of Theorem 5.2 for simple formulas. Suppose $k \ge 2$, and F is a simple formula of size at least $3k$. Applying Lemma 4.1 to F, we obtain sets *Interior* and *Boundary*, simple extended formulas $G_\alpha(y)$ and operations $*_\alpha$ for each $\alpha \in$ *Interior*, and simple formulas G_β for all $\beta \in$ *Boundary* with the properties described in the lemma. By Lemma 7.1 each extended formula $G_\alpha(y)$ is equivalent to either $A_\alpha \times y + B_\alpha$ or $A_\alpha \times y - B_\alpha$, where A_α and B_α are division-free, B_α is simple, $|A_\alpha|, |B_\alpha| \le |G_\alpha(y)|$, and A_α is balanced. Since it is not necessary to apply the construction recursively to A_α for any $\alpha \in$ *Interior*, and

$$\sum_{\alpha\in Interior}|A_\alpha| \le |F| \qquad \text{and} \qquad \sum_{\alpha\in Interior}|B_\alpha| \le |F|,$$

we obtain a sequence of tuples

$$\langle \Delta_i, E_i(W_\delta)_{\delta\in\Delta_i}, \{W_\delta : \delta \in \Delta_i\}\rangle$$

such that $E_i(W_\delta)_{\delta\in\Delta_i} \equiv F$, the depth of $E_i(w_\delta)_{\delta\in\Delta_i}$ is at most $\log|F|$ greater than as stated in the proof of Theorem 5.2 (since this formula now includes the balanced formulas A_α for $\alpha \in$ *Interior* rather than corresponding auxiliary variables a_α), $|W_\delta| \le \lceil k^{-i}|F|\rceil + 2$ for all $\delta \in \Delta_i$, and $|E_i(W_\delta)|_{\delta\in\Delta_i} \le (i+1)|F|$, so that the size of this reconstructed formula grows linearly instead of exponentially with i. Setting $i = \lfloor\log_k|F|\rfloor$, as usual, we obtain the following result.

THEOREM 7.3. *For any simple formula F of size S and any integer $k \ge 2$ there exists a division-free formula G (not necessarily simple) such that*

(i) $G \equiv F$,

(ii) $\text{depth}(G) \le (3k/\log k)\log S + (3k-1)$,

(iii) $|G| \le S(\lfloor\log_k S\rfloor + 1)$.

Kosaraju's results and Theorem 7.2 leave very little room for size–depth tradeoffs for simple formulas. Still, Theorem 7.3 can be applied to prove something new.

COROLLARY 7.4. *Over any field $\mathcal{F}$, for any fixed $\epsilon > 0$ and any simple formula of size S with operations from $\{+, -, \times\} \cup \mathcal{F}$, there is an equivalent division-free formula with*

(i) *depth* $O((\log S)^{1+\epsilon})$ *and size* $O(\frac{S\log S}{\log\log S})$,

as well as an equivalent formula with

(ii) *depth* $O(\log S \log\log S)$ *and size* $O(\frac{S\log S}{\log\log\log S})$.

Proof. Apply Theorem 7.3, setting $k = (\log S)^\epsilon$ in the first case and, in the second case, $k = (\log\log S)(\log\log\log S)$. □

8. Specific formulas and known lower bounds. As mentioned in §1, parallel algorithms for the formula evaluation problem can be modified to transform formulas into small-depth circuits, which can in turn be transformed into formulas of the same depth. We conclude with an example illustrating that polynomial size blowup can arise from this approach, even if one is restricted to division-free formulas. In particular, we shall exhibit a formula of size n such that when the formula evaluation algorithm of Miller and Reif [9] is applied to it, the resulting formula is of size $\Omega(n^{1+\delta})$ for a fixed $\delta > 0$.

For each n, define the formula $F_n(x_1, x_2, \ldots, x_{2n+1})$ as

$$F_n(x_1, x_2, \ldots, x_{2n+1}) = (\cdots(((x_1 \times x_2) + x_3) \times x_4) + \cdots \times x_{2n}) + x_{2n+1}.$$

Clearly, $\text{depth}(F_n(x_1, \ldots, x_{2n+1})) = 2n$ and $|F_n(x_1, \ldots, x_{2n+1})| = 2n + 1$.

Commentz-Walter [4] shows that, over the Boolean semiring $(\{0, 1\}, \wedge, \vee)$ (where negations are disallowed), there are formulas equivalent to $F_n(x_1, \ldots, x_{2n+1})$ with depth $O(\log n)$, but all such formulas have size $\Omega(n \log n)$. Commentz-Walter and Sattler [5] also show that, even if negations can be introduced, any formula of depth $O(\log n)$ that computes $F_n(x_1, \ldots, x_{2n+1})$ must have size $\Omega(\frac{n \log\log n}{\log\log\log n})$. (This nonmonotonic lower bound does not apply if $\oplus$ operations can be introduced.)

Now consider the formula $G_n^{(k)}$, where

$$G_n^{(1)}(x_1, \ldots, x_{2n+1}) = F_n(x_1, \ldots, x_{2n+1})$$

for $F_n(x_1, x_2, \ldots, x_{2n+1})$ as above, and

$$G_n^{(k)}(x_1, \ldots, x_{(2n+1)^k}) = F(G_n^{(k-1)}(x_1^{(k-1)}), \ldots, G_n^{(k-1)}(x_{2n+1}^{(k-1)}))$$

for $k > 1$, where $x_i^{(k-1)} = (x_{(i-1)(2n+1)^{k-1}+1}, \ldots, x_{i(2n+1)^{k-1}})$, $i = 1, \ldots, 2n + 1$.

For $n = 3$ the method of Miller and Reif balances $G_3^{(1)}(x_1, \ldots, x_7)$ to the formula $(((x_1 \times x_2) + x_3) \times (x_4 \times x_6)) + ((x_5 \times x_6) + x_7)$, which induces one more occurrence of the variable x_6. When we try to balance $G_3^{(k)}$ using Miller and Reif's method we induce two copies of each $G_3^{(i)}(x_6^{(i)})$ in each level of the formula $G_3^{(k)}$. This implies that if the balanced formula is of size $S(N)$ for $N = 7^k$, then

$$S(N) = 8S(N/7),$$

which implies that Miller and Reif's method gives a formula of size $n^{\log 8/\log 7} > n^{1.0686}$.

Acknowledgments. We thank Allan Borodin for pointing out the work of Commentz-Walter [4] and Commentz-Walter and Sattler [5]. We also thank the referees for many helpful suggestions and comments.

REFERENCES

[1] R. P. Brent, *The parallel evaluation of general arithmetic expressions*, J. Assoc. Comput. Mach., 21 (1974), pp. 201–206.

[2] S. R. Buss, *The Boolean formula value problem is in ALOGTIME*, in Proc. 19th ACM Symposium on Theory of Computing, New York, NY, ACM Press, 1987, pp. 123–131.

[3] S. Buss, S. Cook, A. Gupta, and V. Ramachandran, *An optimal parallel algorithm for formula evaluation*, SIAM J. Comput., 21 (1992), pp. 755–780.

[4] B. Commentz-Walter, *Size–depth tradeoff in monotone Boolean formulae*, Acta Inform., 12 (1979), pp. 227–243.

[5] B. Commentz-Walter and J. Sattler, *Size–depth tradeoff in non-monotone Boolean formulae*, Acta Inform., 14 (1980), pp. 257–269.

[6] A. GUPTA, *A fast parallel algorithm for recognition of parenthesis languages*, M.S. thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 1985.

[7] S. A. KOSARAJU, *Parallel evaluation of division-free arithmetic expressions*, in Proc. 18th ACM Symposium on Theory of Computing, Berkeley, CA, ACM Press, 1986, pp. 231–239.

[8] S. A. KOSARAJU AND A. L. DELCHER, *A tree partitioning technique with applications to expression evaluation and term matching*, in Proc. 31st IEEE Symposium on Foundations of Computer Science, St. Louis, MO, IEEE Computer Society Press, 1990, pp. 163–172.

[9] G. L. MILLER AND J. REIF, *Parallel tree contraction and its application*, in Proc. 26th IEEE Symposium on Foundations of Computer Science, Portland, OR, IEEE Computer Society Press, 1985, pp. 478–489.

[10] D. E. MULLER AND F. P. PREPARATA, *Restructuring of arithmetic expressions for parallel evaluation*, J. Assoc. Comput. Mach., 23 (1976), pp. 534–543.

SIAM J. COMPUT.
Vol. 24, No. 4, pp. 706–735, August 1995

003

LEARNING ARITHMETIC READ-ONCE FORMULAS*

NADER H. BSHOUTY†, THOMAS R. HANCOCK‡, AND LISA HELLERSTEIN§

Abstract. A formula is read-once if each variable appears at most once in it. An arithmetic read-once formula is one in which the operators are addition, subtraction, multiplication, and division. We present polynomial time algorithms for exact learning of arithmetic read-once formulas over a field. We present a membership and equivalence query algorithm that identifies arithmetic read-once formulas over an arbitrary field. We present a randomized membership query algorithm (i.e., a randomized black box interpolation algorithm) that identifies such formulas over finite fields with at least $2n + 5$ elements (where n is the number of variables) and over infinite fields. We also show the existence of nonuniform deterministic membership query algorithms for arbitrary read-once formulas over fields of characteristic 0, and division-free read-once formulas over fields that have at least $2n^3 + 1$ elements. For our algorithms, we assume we are able to perform efficiently arithmetic operations on field elements and compute square roots in the field. It is shown that the ability to compute square roots is necessary in the sense that the problem of computing $n - 1$ square roots in a field can be reduced to the problem of identifying an arithmetic formula over n variables in that field. Our equivalence queries are of a slightly nonstandard form, in which counterexamples are required not to be inputs on which the formula evaluates to 0/0. This assumption is shown to be necessary for fields of size $o(n/\log n)$ in the sense that we prove there exists no polynomial time identification algorithm that uses only membership and standard equivalence queries.

Key words. learning theory, interpolation, exact identification, polynomials, rational functions, read-once formulas

AMS subject classifications. 41A05, 41A20, 68Q20, 68T05

1. Introduction.

We consider the problem of exactly identifying an unknown formula via oracle queries. In the classical black box interpolation model, there is a black box oracle that computes the unknown target formula, and one is free to substitute inputs into the black box with the goal of constructing a formula that is equivalent to the unknown target. These substitutions are sometimes called membership queries, a term that was developed in the context of boolean functions. Each boolean function corresponds to a subset of its domain (the set of elements for which the output of the function is 1), and thus substitution is equivalent to testing membership in the set.

Another way to acquire information about an unknown formula is via an "equivalence query." In this type of query, one proposes a candidate formula h and asks whether it is equivalent to the unknown target f. If h is equivalent to f, the answer to the query is "yes." If h is not equivalent, the answer is a *counterexample*—an element of the domain on which the outputs of h and f differ. Equivalence queries are motivated in part by the problem of learning from random examples (i.e., from a sequence of random elements of the domain of f, each labeled according to the output of f on that element). Given a long sequence of random examples labeled according to f, one can simulate an equivalence query by testing the hypothesis h on those examples. If h disagrees with some example, that example is a counterexample, otherwise h is at least a good approximation for f (with high probability).

*Received by the editors August 20, 1992; accepted for publication (in revised form) March 10, 1994.

†Department of Computer Science, University of Calgary, 2500 University Drive N.W., Calgary, Alberta, Canada T2N 1N4 (bshouty@cpsc.ucalgary.ca). This research was supported in part by the National Sciences and Engineering Research Council of Canada.

‡Siemens Corporate Research, 755 College Road East, Princeton, New Jersey 08540 (hancock@learning.scr.siemens.com). This work was done while the author was a graduate student at Harvard University, supported by Office of Naval Research grant N00014-85-K-0445 and National Science Foundation grant NSF-CCR-89-02500.

§Department of Electrical Engineering and Computer Science, Northwestern University, 2145 Sheridan Road, Evanston, Illinois 60208-3118 (hstein@eecs.nwu.edu).

The model of exact learning with membership and equivalence queries was introduced by Angluin [1] and has been the subject of much research in the learning theory community (we give a more precise definition later).

A multivariate formula is read-once if each variable appears at most once in it. Angluin, Hellerstein, and Karpinski [2] proved that boolean read-once formulas over the basis (AND, OR, NOT) can be exactly identified in polynomial time using membership and equivalence queries (this is not possible using either type of query exclusively). This result has been generalized to include other classes of boolean read-once formulas [10], [6], [12], [5].

We study the problem of exactly learning arithmetic read-once formulas. These are formulas over a field where the basis functions are arithmetic operations (addition, subtraction, multiplication, and division) over that field. We give an efficient deterministic algorithm for formulas over arbitrary fields using membership and equivalence queries. We further show that membership queries alone suffice (if one allows randomization) for fields that have at least $2n + 5$ elements, where n is the number of variables. We also show nonconstructively that membership queries alone suffice for deterministic (but nonuniform) identification of formulas over fields of characteristic 0 or division-free formulas over fields of at least $2n^3 + 1$ elements.

The membership query only results can be rephrased as interpolation results using black box substitutions. Arithmetic read-once formulas over a field compute a subclass of the multivariate rational functions over that field. Division-free arithmetic read-once formulas compute a subclass of the polynomial functions. The learnability of sparse polynomials and rational functions over fields using membership and enhanced membership queries has been previously studied [9], [8], [3], [15], [4]. The classes of sparse polynomials and rational functions are incomparable to the class of arithmetic read-once formulas; sparse polynomials and rational functions are not necessarily read-once, and the polynomials obtained from expanding division-free read-once formulas are not generally sparse. The results in this paper are the first nontrivial polynomial interpolation results for a class of nonsparse rational functions. The special case of division-free read-once formulas with sparse polynomial expansions was studied by Lhotzky [14].

We present a single core algorithm that employs new algebraic techniques for exact identification of an arithmetic read-once formula over any field, using membership queries (or equivalently, substitutions). The algorithm requires a set of "justifying assignments" (input settings to the variables that satisfy certain properties defined below) as additional input. This algorithm relies on being able to compute efficiently the arithmetic functions on field elements, and also on being able to compute square roots in the field. The ability to compute square roots is shown as necessary for identifying this class, since we are able to reduce the problem of computing $n - 1$ square roots in a field to that of identifying an n variable arithmetic read-once formula over that field. The upper bounds we give in this paper are based on unit time computation of square roots and field operations.

We present several alternate methods for finding justifying assignments. If the field is sufficiently large (at least $2n + 5$ elements), we can use randomized membership queries. We also prove (nonconstructively) that a nonuniform deterministic membership query algorithm exists if a formula is division-free, or if the field has characteristic 0 (e.g., the reals). In the latter case we use a result of Heintz and Schnorr [13]. Membership queries alone do not provide enough information to identify arithmetic read-once formulas over small finite fields, so to handle an an arbitrary field we present a technique that uses equivalence queries as well as membership queries. These equivalence queries are slightly nonstandard in that we add a minor restriction on what counterexample may be returned. In particular, if there exists a counterexample on which the target formula does not evaluate to $0/0$, we require that such a counterexample be returned. This assumption is shown to be necessary for fields of size

$o(n/\log n)$. For fields of this size it is shown that there is no polynomial time identification algorithm that uses just membership and standard equivalence queries.

Other work related to this paper is by Goldman, Kearns, and Schapire [7] who use nonadaptive randomized membership queries to identify restricted classes of boolean read-once formulas. They also show nonconstructively the existence of deterministic algorithms.

In §2 we present our definitions and basic notation for this paper. In §3 we discuss the core algorithm, and in §4 we describe the techniques for obtaining justifying assignments and state our positive results as theorems. Section 5 describes the lower bounds. We conclude in §6 with a table summarizing our results.

This paper uses a number of basic facts from linear algebra. As an aid to readers without a strong background in this area we include many of these facts as propositions, either without proof or with the proofs deferred to the appendix.

2. Definitions and notation. A *formula* is a rooted tree whose leaves are labeled with variables or constants from some domain, and whose internal nodes or *gates* are labeled with elements from a set of *basis* functions over that domain. A *read-once formula* is a formula for which no variable appears on two different leaves. An *arithmetic read-once formula* over a field $\mathcal{K}$ is a read-once formula over the basis of addition, subtraction, multiplication, and division of field elements, whose leaves are labeled with variables or constants from $\mathcal{K}$.

For notational convenience we define a modified basis and consider our arithmetic read-once formulas defined over this basis. Let $\mathcal{K}$ be an arbitrary field. Our modified basis for arithmetic read-once formulas over $\mathcal{K}$ will include only two nonunary functions, addition $(+)$ and multiplication $(\times)$. The unary functions in the basis are $(ax + b)/(cx + d)$ for every $a, b, c, d \in \mathcal{K}$ such that $ad - bc \neq 0$ (this requirement prevents $ax + b$ and $cx + d$ from being identically 0 or differing by just a constant factor). We also assume that nonconstant formulas over this modified basis do not contain constants in their leaves.[1]

We represent such a unary function as f_A, where

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}.$$

The restriction on a, b, c, and d is equivalent to saying that the determinant of A (denoted $\det(A)$) is nonzero. This representation becomes useful when we think of the column vector $(a\ \ b)^T$ as representing the field element a/b. With this representation we may compute $f_A(x)$ by multiplying $A(x\ \ 1)^T$.

The value of a read-once formula on an assignment to its variables is determined by evaluating the formula bottom up. This raises the issue of division by zero. We handle this by defining our basis functions over the extended domain $\mathcal{K} \cup \{\infty, \text{ERROR}\}$, where ∞ represents $1/0$ and ERROR represents $0/0$. Note that $(a\ \ b)^T$ now corresponds to a domain element for any choice of $a, b \in \mathcal{K}$, since if $b = 0$ then a/b is either ∞ or ERROR depending on a. On field elements the basis functions are defined in the obvious way. For the special values we define our basis function as follows (assume $x \in \mathcal{K} - \{0\}$, $y \in \mathcal{K} \cup \{\infty, \text{ERROR}\}$, and A is as above):

[1] There are arithmetic read-once formulas such as $x/0$ and $0/x$ for which there is no equivalent read-once formula over the modified basis having no constants in the leaves. However, such formulas are in some sense degenerate. For example, note that $x/0$ is algebraically undefined and $0/x$ would normally be reduced to 0 even though $0/x$ does not evaluate to 0 at the point $x = 0$. We shall ignore these degenerate formulas and define the class of arithmetic read-once formulas as precisely those that are constant or for which there is an equivalent read-once formula over the modified basis with no constants in the leaves.

$$y + \text{ERROR} = y \times \text{ERROR} = f_A(\text{ERROR}) = \text{ERROR},$$
$$x + \infty = x \times \infty = \infty,$$
$$0 + \infty = \infty \times \infty = \infty,$$
$$0 \times \infty = \infty + \infty = \text{ERROR},$$
$$f_A(\infty) = \begin{cases} \frac{a}{c} & c \neq 0, \\ \infty & c = 0, \end{cases} \quad \text{and} \quad f_A\left(\tfrac{-d}{c}\right) = \infty \text{ if } c \neq 0.$$

Note that $f_A(\infty)$ is represented by $A(1\ \ 0)^T$. By Property 2.2 in §2.3 these definitions are designed so that the output of the read-once formula is the same as it would be if the formula was first expanded and simplified to be in the form $p(x_1, \ldots, x_n)/q(x_1, \ldots, x_n)$ for some polynomials p and q, where $\gcd(p, q) = 1$, and then evaluated.

The distinction between ∞ and ERROR is an important one. The value ∞ is essentially just another domain value (although we make no membership queries with variables set to ∞). Introducing ∞ means that our unary basis functions are bijections from $\mathcal{K} \cup \{\infty\}$ to $\mathcal{K} \cup \{\infty\}$. It is possible for subtrees of a formula to evaluate to ∞ but for the entire tree to evaluate to a value from $\mathcal{K}$. This is not the case for ERROR, which, once it appears anywhere within a formula, is necessarily propagated to the root.

We say that a formula f is *defined* on the variable set V if all variables appearing in f are members of V. Let $V = \{x_1, \ldots, x_n\}$. We say a formula f *depends* on variable x_i if there are values $x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)}$ and $x_i^{(1)}$ in $\mathcal{K}$ for which

$$f(x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)}) \neq f(x_1^{(0)}, \ldots, x_{i-1}^{(0)}, x_i^{(1)}, x_{i+1}^{(0)}, \ldots, x_n^{(0)}),$$

and both those values of f are not ERROR. We call such an input vector $v = (x_1^{(0)}, \ldots, x_n^{(0)})$ a *justifying assignment* for x_i (this is a slight modification of the definition in previous literature to account for the ERROR possibility).

An assignment of values to some subset of a read-once formula's variables defines a *projection*, which is the formula obtained by hard-wiring those assigned variables to their values in the formula and then rewriting the formula to eliminate constants from the leaves. Let $f|(x \leftarrow x^{(0)})$ denote the projection of f obtained by hard-wiring x to the value $x^{(0)}$. For a set of variables $W \subseteq V$ and an input setting $v \in \mathcal{K}^n$, let $f|(W \leftarrow v)$ denote the projection of f obtained by hard-wiring each $x \in W$ to its value in v.

If f depends on variable x_1, we say a value $x_2^{(0)} \in \mathcal{K} \cup \{\infty\}$ for x_2 *blocks* x_1 in f if the projection $f|(x_2 \leftarrow x_2^{(0)})$ no longer depends on x_1.

A justifying assignment for a variable gives us values to which we can set the remaining variables such that the induced projection depends on that single variable. We are also interested in input settings that fix all but two or three variables so that the induced projection depends on those two or three variables. We call such settings *two-* and *three*-justifying assignments, respectively.

For any pair of variables x_i and x_j that appear in a read-once formula, there is a unique node farthest from the root that is an ancestor of both x_i and x_j, called their *lowest common ancestor*, which we write as $\text{lca}(x_i, x_j)$. We shall refer to the *type* of $\text{lca}(x_i, x_j)$ as the basis function computed at that gate. We say that a set W of variables has a common lca if there is a single node that is the lca of every pair of variables in W.

We define the *skeleton* of a formula f as the tree obtained by deleting any unary gates in f and removing the labels from any remaining internal nodes (i.e., the skeleton describes the parenthesization of an expression, but not the actual operations or embedded constants).

2.1. Identification with queries. The learning criterion we consider is *exact identification*. There is a formula f called the *target formula*, which is a member of a class of formulas

C defined over the variable set V. The goal of the learning algorithm is to halt and output a formula h from C that is equivalent to f.

In a *membership query*, the learning algorithm supplies values $(x_1^{(0)},\ldots,x_n^{(0)})$ for the variables in V as input to a *membership oracle* and receives in return the value of $f(x_1^{(0)},\ldots,x_n^{(0)})$. Note that if f' is a projection of f, it is possible to simulate a membership oracle for f' using a membership oracle for f.

In an *equivalence query*, the learning algorithm supplies a candidate read-once formula h as input to an *equivalence oracle*, and the reply of the oracle is either "yes," signifying that h is equivalent to f, or a *counterexample*, which is an input setting $v = (x_1^{(0)}, \ldots, x_n^{(0)})$ such that $h(v) \neq f(v)$. In the standard model the choice of the counterexample v is arbitrary. In this paper we consider a slightly nonstandard model in which the counterexample v is arbitrary, except that it will not have $f(v) =$ ERROR unless no other counterexamples are available.

A technical detail is how much time to charge for making a query. We follow Angluin, Hellerstein, and Karpinski [2] and charge for both setting up the query and invoking the oracle. In a membership query consisting of an assignment to n variables, we charge unit time for specifying each of the n assignments. Therefore, the set-up cost of a membership query is is typically $O(n)$ (it can be lower if the query is formed by changing only a small number of bits in the previous query, as is the case in several of our algorithms). The setup cost of an equivalence query involving a read-once formula is also typically $O(n)$ because we charge according to the number of nodes in the input formula. We charge unit time for invoking either the membership oracle or the equivalence oracle once the query is set up.

2.2. Properties of unary functions. Here we list some basic properties of unary functions f_A. These show some of the advantages of the matrix notation. In all of the following we assume

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}.$$

For the basis functions A is nonsingular. When we consider projections that assign values to all but one variable in a read-once formula, the induced function can be constant or have a degenerate form such as $0/x$. These correspond to functions f_A for which $\det(A) = 0$.

Using basic linear algebra, we establish the following properties of our representation. The proofs are technical, though straightforward, and we defer them to the appendix.

PROPERTY 2.1.

(1) *The function f_A is a bijection from $\mathcal{K} \cup \{\infty\}$ to $\mathcal{K} \cup \{\infty\}$ if and only if* $\det(A) \neq 0$. *Otherwise, f_A is either a constant value from $\mathcal{K} \cup \{\infty,$* ERROR*$\}$ or else a constant value from $\mathcal{K} \cup \{\infty\}$, except on one input value on which it is* ERROR.

(2) *The functions f_A and $f_{\lambda A}$ are equivalent for any $\lambda \neq 0$.*

(3) *Given any three distinct points $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, and $p_3 = (x_3, y_3)$,*

(a) *if p_1, p_2, p_3 are on a line then there exists a unique function f_A with $f_A(x) = ax + b$ that satisfies $f_A(x_1) = y_1$, $f_A(x_2) = y_2$, and $f_A(x_3) = y_3$;*

(b) *if p_1, p_2, p_3 are not on a line then there exists a unique function f_A with* $\det(A) \neq 0$ *that satisfies $f_A(x_1) = y_1$, $f_A(x_2) = y_2$, and $f_A(x_3) = y_3$.*

(4) *If functions f_A and f_B are equivalent and* $\det(A), \det(B) \neq 0$, *then there is a constant λ for which $\lambda A = B$.*

(5) *The functions $(f_A \circ f_B)$ and f_{AB} are equivalent.*

(6) *If* $\det(A) \neq 0$, *functions f_A^{-1} and $f_{A^{-1}}$ are equivalent.*

(7) $f_A(\lambda x) = f_{A\left(\begin{smallmatrix}\lambda & 0\\ 0 & 1\end{smallmatrix}\right)}(x)$ *and* $f_A(\lambda + x) = f_{A\left(\begin{smallmatrix}1 & \lambda\\ 0 & 1\end{smallmatrix}\right)}(x)$. $\lambda f_A(x) = f_{\left(\begin{smallmatrix}\lambda & 0\\ 0 & 1\end{smallmatrix}\right)A}(x)$ *and* $\lambda + f_A(x) = f_{\left(\begin{smallmatrix}1 & \lambda\\ 0 & 1\end{smallmatrix}\right)A}(x)$.

2.3. Properties of read-once formulas. In this section we state some important properties of read-once formulas, which we shall frequently use in the subsequent sections. The proofs are technical and we defer them to the appendix.

Property 2.2 shows that the definitions at the start of §2 for evaluating an arithmetic read-once formula indeed do the right thing. We show that the output of the formula is the same as it would be if we first simplified the formula to the form p/q for two polynomials p and q with $\gcd(p, q) = 1$, and then evaluated it.

First we give an inductive definition of what it means for an arithmetic read-once formula to compute a rational formula (by the natural means of expanding and then simplifying the formula). We say an arithmetic read-once formula f *computes* the rational function p/q defined as follows: If f is a single leaf labeled with a variable x_1 we say f computes x_1, and if f is a constant a we say f computes a. If the root of f is a unary function f_A with $A = \left(\begin{smallmatrix}a & b\\ c & d\end{smallmatrix}\right)$, whose input is a subformula computing p_1/q_1, we say that f computes $(ap_1 + bq_1)/(cp_1 + dq_1)$. If the root of f is a $\times$ gate whose two inputs (w.l.o.g.) compute p_1/q_1 and p_2/q_2, we say that f computes $(p_1p_2)/(q_1q_2)$. If the root of f is a $+$ gate whose two inputs (without loss of generality (w.l.o.g.)) compute p_1/q_1 and p_2/q_2, we say that f computes $(p_1q_2 + p_2q_1)/(q_1q_2)$. Furthermore, if f computes p/q and $\gcd(p, q) = r$, we also say that f computes $(p/r)/(q/r)$.

PROPERTY 2.2. *Let* $f(x_1, \ldots, x_n)$ *be an arithmetic read-once formula that computes*

$$p(x_1, \ldots, x_n)/q(x_1, \ldots, x_n),$$

where $\gcd(p, q) = 1$. *Then for any* $v = (x_1^{(0)}, \ldots, x_n^{(0)})$,

(1) $f(v) = a \notin \{\infty, 0\}$ *if and only if* $p(v)/q(v) = a$;

(2) $f(v) = 0$ *if and only if* $p(v) = 0$ *and* $q(v) \neq 0$;

(3) $f(v) = \infty$ *if and only if* $p(v) \neq 0$ *and* $q(v) = 0$;

(4) $f(v) = \text{ERROR}$ *if and only if* $p(v) = q(v) = 0$.

Note that this property is not true of read-twice formulas (e.g., x/x, which by our definition computes 1, fails condition (4)).

Property 2.3 implies that the type of the lca of two variables in an arithmetic read-once formula is unique (i.e., there are no two equivalent read-once formulas in which the same pair of variables have different lca types).

PROPERTY 2.3. *There exist no nonsingular matrices* $A, B, C, D, E,$ *and* F *such that*

$$f_A(f_B(x_1) \times f_C(x_2)) \equiv f_D(f_E(x_1) + f_F(x_2)).$$

Property 2.4 states that an arithmetic read-once formula with two inputs is a representation that is unique except for fairly minor variations of the unary functions (e.g., corresponding to whether a constant multiplicative factor is applied to the output of a gate or to its input(s)).

PROPERTY 2.4. *For nonsingular matrices* A_i, B_i, *and* C_i $(i = 1, 2)$ *we have*

(1) $f_{C_1}(f_{A_1}(x_1) \times f_{B_1}(x_2)) = f_{C_2}(f_{A_2}(x_1) \times f_{B_2}(x_2))$ *if and only if*

$$f_{A_2}(x) = \alpha f_{A_1}(x), \quad f_{B_2}(x) = \beta f_{B_1}(x) \text{ and } f_{C_2}(x) = f_{C_1}(\gamma x)$$

or

$$f_{A_2}(x) = \frac{\alpha}{f_{A_1}(x)}, \quad f_{B_2}(x) = \frac{\beta}{f_{B_1}(x)}, \quad \text{and} \quad f_{C_2}(x) = f_{C_1}\left(\frac{1}{\gamma x}\right)$$

for some constants α, β, *and* γ, *where* $\alpha\beta\gamma = 1$;

(2) $f_{C_1}(f_{A_1}(x_1) + f_{B_1}(x_2)) = f_{C_2}(f_{A_2}(x_1) + f_{B_2}(x_2))$ *if and only if*

$$f_{A_2}(x) = \alpha + \eta f_{A_1}(x), \quad f_{B_2}(x) = \beta + \eta f_{B_1}(x), \quad \text{and} \quad f_{C_2}(x) = f_{C_1}\left(\frac{1}{\eta}(\gamma + x)\right)$$

for some constants α, β, *and* γ, *where* $\alpha + \beta + \gamma = 0$ *and some nonzero constant* η.

Property 2.5 addresses the question of when adjacent + or × gates in a formula may be collapsed together to form a single gate (of greater fan-in). Because there may be an intervening unary function, this is not always possible (unlike the situation for boolean formulas, where after pushing negations to the leaves, adjacent AND or OR gates can be merged). It turns out that such a collapse is possible only when there is either no intervening unary function, or else the intervening function (f_B below) is of a fairly simple form (such as simply multiplying its input by a constant).

PROPERTY 2.5. *For nonsingular matrices* A_i $(i = 1, 2, 3)$, B, *and* C,

(1) *there exists matrices* A_i' *and* C *such that*

$$f_C(f_{A_3}(x_3) \times f_B(f_{A_1}(x_1) \times f_{A_2}(x_2))) = f_{C'}(f_{A_3'}(x_3) \times f_{A_1'}(x_1) \times f_{A_2'}(x_2))$$

if and only if

$$B = \begin{pmatrix} \alpha_1 & 0 \\ 0 & \alpha_2 \end{pmatrix} \text{ or } B = \begin{pmatrix} 0 & \alpha_1 \\ \alpha_2 & 0 \end{pmatrix}$$

for some nonzero constants α_1 *and* α_2;

(2) *there exists matrices* A_i' *and* C *such that*

$$f_C(f_{A_3}(x_3) + f_B(f_{A_1}(x_1) + f_{A_2}(x_2))) = f_{C'}(f_{A_3'}(x_3) + f_{A_1'}(x_1) + f_{A_2'}(x_2))$$

if and only if

$$B = \begin{pmatrix} \alpha_1 & \alpha_3 \\ 0 & \alpha_2 \end{pmatrix}$$

for some nonzero constants α_1 *and* α_2 *and a constant* α_3.

3. The core algorithm. Our results for learning arithmetic read-once formulas are all based on the following general purpose core algorithm, which uses deterministic membership queries and can be applied to learn read-once formulas over any field. The core algorithm takes as input three-justifying assignments for each subset of three variables in a target arithmetic read-once formula f, and returns an equivalent arithmetic read-once formula. In this section we assume that the justifying assignments are already available. In subsequent sections, we discuss different techniques for obtaining such justifying assignments, depending on the query model and field in question.

In our discussion of the core algorithm, we assume the field over which the formula is defined has at least three elements, since division is not an interesting operation in two element fields (the division-free case for two element fields is covered in other papers [5], [11]).

The algorithm is based on the reduction of Lemma 3.2 presented in §3.1. The reduction transforms our problem to the problem of finding polynomial time routines that (1) learn the

skeleton of the target formula using membership queries and justifying assignments, and (2) learn read-once formulas that contain at most one nonunary gate using membership queries and justifying assignments.

In Lemmas 3.3 and 3.4 we reduce the problem of skeleton construction (problem (1) above) to the two subtasks of (1a) determining the type of the lca ($\times$ or $+$) of each pair of variables and (1b) determining (in some cases) which two out of three variables have the deeper lca when all the pairwise lca's are of the same type.

These techniques are taken or generalized from previous work on boolean read-once formulas. In §§3.2, 3.3, and 3.4 we present the key new results for the arithmetic read-once formula problem that solve problems (1a), (1b), and (2), respectively.

The resulting algorithm allows us to prove the following result in a computational model that allows unit time computation of field operations and square roots.

LEMMA 3.1. *There is a polynomial time algorithm that uses membership queries and three-justifying assignments to exactly identify an arithmetic read-once formula over an arbitrary field. This algorithm requires $O(n^4)$ time, $O(n^3)$ membership queries, and $n-1$ square root computations.*

3.1. General techniques for finding and using the skeleton.

LEMMA 3.2. *Given the skeleton of an arithmetic formula f as well as two-justifying assignments for each pair of variables, the problem of polynomial time exact identification of f with membership queries is polynomial time reducible to that of identifying a formula that has a single nonunary gate with membership queries and justifying assignments.*

Proof. We find two variables x_i and x_j that are siblings in the skeleton. Using a two-justifying assignment $a = (a_1, \ldots, a_n)$ for x_i and x_j, we have

$$f(a_1, \ldots, a_{i-1}, x_i, a_{i+1}, \ldots, a_{j-1}, x_j, a_{j+1}, \ldots, a_n) = f_C(f_A(x_i)\ op\ f_B(x_j)),$$

where $op \in \{+, \times\}$. Given that we can identify this one-gate formula (i.e., find A, B, C, and op within the factors allowed in Property 2.4), we now reduce the problem to exact identification of an arithmetic read-once formula with $n-1$ variables as follows: We will substitute

$$y = f_A(x_i)\ op\ f_B(x_j) = f_{D_{x_i}B}(x_j),$$

where

$$D_{x_i} = \begin{pmatrix} f_A(x_i) & 0 \\ 0 & 1 \end{pmatrix}$$

for $op = \times$ and

$$D_{x_i} = \begin{pmatrix} 1 & f_A(x_i) \\ 0 & 1 \end{pmatrix}$$

for $op = +$ (this is by Property 2.1 (7)).The new read-once formula f' obtained from this substitution is over the $n-1$ variables $(y, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n)$. Now, to simulate the membership query

$$f'(y^{(0)}, x_1^{(0)}, \ldots, x_{i-1}^{(0)}, x_{i+1}^{(0)}, \ldots, x_{j-1}^{(0)}, x_{j+1}^{(0)}, \ldots, x_n^{(0)})$$

we ask the membership query

$$f(x_1^{(0)}, \ldots, x_{i-1}^{(0)}, a_i, x_{i+1}^{(0)}, \ldots, x_{j-1}^{(0)}, f_{B^{-1}D_{a_i}^{-1}}(y^{(0)}), x_{j+1}^{(0)}, \ldots, x_n^{(0)}).$$

Let J be a set of justifying assignments for each set of two variables in f. Then it is easy to see that

$$J' = \{(f_{D_{b_i}B}(b_j), b_1, \ldots, b_{i-1}, b_{i+1}, \ldots, b_{j-1}, b_{j+1}, \ldots, b_n) | (b_1, \ldots, b_n) \in J\}$$

is a set of justifying assignments for each set of two variables in f'. We repeat this process $n-1$ times until we have identified f. □

Define the *metaskeleton* of an arithmetic read-once formula f as the graph obtained from the skeleton of f by merging together all adjacent nodes labeled with the same type of gate ($\times$ or $+$). The following lemma is proved by Angluin, Hellerstein, and Karpinski [2]. (Their proof is for boolean formulas over {AND, OR}, but the same proof works for $\{+, \times\}$.)

LEMMA 3.3 [2]. *There is a polynomial time algorithm for finding the metaskeleton of f given the type of the lca of each pair of variables.*

Given the metaskeleton, the remaining problem for skeleton construction is to reconstruct those portions where all the gates are of the same type (corresponding to the nodes of the metaskeleton). To do this we shall perform a procedure (described in §3.3) in which we examine sets of three variables and try to determine which two of them (if any) have the deeper lca. We can make this determination in some but not all cases. However, our procedure does satisfy the following two conditions: (1) every time we decide a particular pair has a deeper lca we are correct, and (2) we always determine the deeper pair in the case where the deeper lca is an immediate child of the shallower one. Lemma 3.4 shows that this gives us enough information to build the skeleton.

LEMMA 3.4. *Let S be a set containing elements of the form $\{\{x_i, x_k\}, x_j\}$. Suppose the following two conditions are true of S for a read-once formula f:*

(1) *If $\{\{x_i, x_k\}, x_j\} \in S$ then $\mathrm{lca}(x_i, x_k)$ is below $\mathrm{lca}(x_i, x_j)$.*

(2) *If $\mathrm{lca}(x_i, x_k)$ is below $\mathrm{lca}(x_i, x_j)$ and there are no intervening nonunary gates then $\{\{x_i, x_k\}, x_j\} \in S$.*

Then there is a polynomial time algorithm for reconstructing the skeleton of f given S.

Proof. For an arbitrary pair of variables x_i and x_j, define the following two sets:

$$W = \{x_k \mid \{\{x_i, x_k\}, x_j\} \in S\},$$

$$U = \{x_l \mid \text{for some } x_k \in W, \{\{x_l, x_k\}, x_j\} \in S\}.$$

We claim that the set $\{x_i\} \cup W \cup U$ consists of exactly those variables that appear in the same subformula of the formula rooted at $\mathrm{lca}(x_i, x_j)$, as does x_i (let us call this subformula $\hat{f}$).

If $x_k \in W$ then condition (1) implies that x_k appears in $\hat{f}$. Then this implies that any x_l that we add to U must also appear in $\hat{f}$. Thus $\{x_i\} \cup W \cup U$ is a subset of the variables appearing in $\hat{f}$.

Let G be the root node of $\hat{f}$ (a child of $\mathrm{lca}(x_i, x_j)$ in f). If x_k appears in $\hat{f}$ and $\mathrm{lca}(x_i, x_k) = G$ then condition (2) implies that $x_k \in W$. If x_l appears in $\hat{f}$ and $\mathrm{lca}(x_i, x_l) \neq G$, then for an x_k with $\mathrm{lca}(x_i, x_k) = G$ (and hence in W), condition (2) implies that $\{\{x_k, x_l\}, x_j\} \in S$, and hence that x_l will be in U. Thus every variable appearing in $\hat{f}$ (besides x_i) is in either U or W. This proves the claim.

The lemma easily follows from the claim since, if the skeleton has more than one gate, we can find an x_i and x_j such that $W \cup U \neq \emptyset$, and then we can partition the variables and learn the two subskeletons recursively. □

3.2. Determining the type of an lca with blocking values. In this section we show how to determine the type of the lca for a pair of variables in the target formula. We consider only $\text{lca}(x_1, x_2)$ for notational convenience. We use the following criterion to determine the type of the lca. (We already know that the type of $\text{lca}(x_i, x_j)$ is unique by Property 2.3, although the correctness of this lemma gives an alternate, more involved proof of that fact.)

LEMMA 3.5. *Suppose* $f(x_1, x_2, \ldots, x_n)$ *is an arithmetic read-once formula over a field, and the projection* $f'(x_1, x_2) = f(x_1, x_2, x_3^{(0)}, \ldots, x_n^{(0)})$ *depends on* x_1 *and* x_2. *The type of* $\text{lca}(x_1, x_2)$ *in* f *is* $\times$ *if and only if* x_2 *has exactly two blocking values for* x_1 *in* f'. *The type of* $\text{lca}(x_1, x_2)$ *in* f *is* $+$ *if and only if* x_2 *has exactly one blocking value for* x_1 *in* f'.

Proof. Suppose $x_3^{(0)}, \ldots, x_n^{(0)} \in \mathcal{K}$ are values such that $f'(x_1, x_2) = f(x_1, x_2, x_3^{(0)}, \ldots, x_n^{(0)})$ depends on x_1 and x_2. Then it must be true that $f'(x_1, x_2) = f_C(f_A(x_1)\ op\ f_B(x_2))$, where $op \in \{\times, +\}$ is the operation computed at $\text{lca}(x_1, x_2)$. Furthermore, matrices A, B, and C must be nonsingular.

If $op = \times$, $f'(x_1, x_2^{(0)})$ depends on x_1 if and only if $f_B(x_2^{(0)}) \neq 0$ or ∞. Thus the distinct values of x_2 that block x_1 in f' are $f_B^{-1}(0)$ and $f_B^{-1}(\infty)$.

If $op = +$, $f'(x_1, x_2^{(0)})$ depends on x_1 if and only if $f_B(x_2^{(0)}) \neq \infty$. Thus the unique value of x_2 that blocks x_1 in f' is $f_B^{-1}(\infty)$. □

Thus, to determine the type of $\text{lca}(x_1, x_2)$ for any pair of variables we need only determine the number of blocking values in $f'(x_1, x_2)$. We first look for two values of x_2 that do not block x_1. We do this by testing three arbitrary field elements. Either two of these values for x_2 block x_1 (in which case we've found that there are two blocking values and are done) or else two of them do not. In the latter case let $x_2^{(1)}$ and $x_2^{(2)}$ be the two nonblocking values. For $i = 1, 2$ define $b_i = f_B(x_2^{(i)})$. We know $b_1 \neq b_2$ (Property 2.1 (1)). Let us define matrices H_i as follows (where op is the type of $\text{lca}(x_1, x_2)$):

$$\begin{aligned} f_{H_i}(x_1) &= f'(x_1, x_2^{(i)}) \\ &= f_C(f_A(x_1)\ op\ f_B(x_2^{(i)})) \\ &= f_C(f_A(x_1)\ op\ b_i). \end{aligned}$$

By substituting three values for x_1 into $f'(x_1, x_2^{(i)})$ (Property 2.1 (3)), we can solve this to find H_1 and H_2 (within a constant factor). Since x_1 is not blocked by $x_2^{(1)}$, we know that matrix H_1 is invertible (Property 2.1 (1)). We define $D = H_1^{-1} H_2$.

LEMMA 3.6. *Matrix* D *has two distinct eigenvalues if* $\text{lca}(x_1, x_2)$ *is multiplication and only one eigenvalue if* $\text{lca}(x_1, x_2)$ *is addition.*

Proof. A value $y \in \mathcal{K} \cup \{\infty\}$ for x_1 blocks x_2 in f' if and only if

$$f'(y, x_2^{(1)}) = f'(y, x_2^{(2)}). \tag{3}$$

Equation (3) is true if and only if $f_{H_1}(y) = f_{H_2}(y)$ (by the definition of H_i), which is true if and only if $y = f_D(y)$ (by Properties 2.1 (6) and 2.1 (5)).

For a value $y \in \mathcal{K}$, $y = f_D(y)$ if and only if $D(y\ \ 1)^T = \lambda(y\ \ 1)^T$ for some $\lambda \in \mathcal{K}$ or, in other words, if and only if λ is an eigenvalue of D and $(y\ \ 1)^T$ is a corresponding eigenvector of D.

For $y = \infty$, $y = f_D(y)$ if and only if $D(1\ \ 0)^T = \lambda(1\ \ 0)^T$ for some $\lambda \in \mathcal{K}$ or, in other words, if and only if λ is an eigenvalue of D and $(1\ \ 0)^T$ is an eigenvector of D.

For each eigenvalue of D there is exactly one eigenvector of the form $(y\ \ 1)^T$ or $(1\ \ 0)^T$ (D cannot be a multiple of I since f_{H_1} and f_{H_2} are not equivalent). The claim follows from Lemma 3.5. □

Thus, to determine the type of $\text{lca}(x_1, x_2)$ we need only compute the number of eigenvalues for matrix D. The eigenvalues are the roots of the quadratic equation $\det(D - \lambda I) = 0$. To

determine whether the equation has one or two roots we need only check whether or not the discriminant is 0 (true for any field $\mathcal{K}$).

Note that finding the eigenvalues requires solving a quadratic equation, and hence taking square roots in the field. We shall later do this to compute one blocking value for each internal node of the formula (in §3.4).

3.3. Building a skeleton when all lca's are the same type. By Lemma 3.3 the previous section allows us to reduce the skeleton construction problem to the case where all the nonunary gates in the formula are the same type ($+$ or $\times$). To solve this problem we use Lemma 3.4. This requires building a set S, and in this section we present a technique to do this. Lemma 3.7 states a criterion for determining whether to add $\{\{x_1, x_2\}, x_3\}$ to S (signifying proof that $\text{lca}(x_1, x_2)$ is below $\text{lca}(x_1, x_3)$). The criterion is to add the element if and only if both conditions (4) and (5) of the lemma fail. Applying this criterion to all three-tuples of variables gives us a set S that satisfies the conditions we need to apply Lemma 3.4.

LEMMA 3.7. *Suppose $x_4^{(0)}, \ldots, x_n^{(0)}$ are the values of $x_4, \ldots, x_n$ in a three-justifying assignment for $\{x_1, x_2, x_3\}$. Let $f'(x_1, x_2, x_3) = f(x_1, x_2, x_3, x_4^{(0)}, \ldots, x_n^{(0)})$ and define the following two conditions:*

(4) *Every value of x_1 that blocks x_2 in f' also blocks x_3 in f'.*

(5) *Every value of x_2 that blocks x_1 in f' also blocks x_3 in f'.*

Then the following two statements are true:

1. *If variables x_1 and x_2 do not have the deepest pairwise* lca *of $\{x_1, x_2, x_3\}$, then either condition* (4) *or condition* (5) *is true.*
2. *If* $\text{lca}(x_1, x_2)$ *is a child of* $\text{lca}(x_1, x_3)$ *in the skeleton of f, expressed with as few nonunary gates as possible, then both conditions* (4) *and* (5) *are false.*

Proof. First we show statement 1. If $\text{lca}(x_1, x_2)$ is not the deepest lca, then either all three variables have the same lca or else x_3 has a deeper lca with one of x_1 or x_2 than with the other. If they share the same lca, then any value of x_1 that blocks x_2 must force an input to that gate to ∞ (or 0 if the gate computes $\times$). Hence that value also blocks x_3 (i.e., condition (4) is true). If $\text{lca}(x_1, x_3)$ is below $\text{lca}(x_1, x_2)$, then any value of x_2 that blocks x_1 must set an input to a node of which x_3 is a descendant to ∞ (or 0 if the node computes $\times$). Hence that value for x_2 also blocks x_1, and condition (5) is true. A symmetric argument applies when $\text{lca}(x_2, x_3)$ is below $\text{lca}(x_1, x_2)$.

Now we prove statement 2. In this case, $f'(x_1, x_2, x_3)$ can be written as

$$f_C(f_{A_3}(x_3)\ op\ f_B(f_{A_1}(x_1)\ op\ f_{A_2}(x_2))).$$

Since $\text{lca}(x_1, x_2)$ is a child of $\text{lca}(x_1, x_3)$ in a read-once formula with as few nonunary gates as possible, we may assume (by Property 2.5) that B is not of the form

$$\begin{pmatrix} \alpha_1 & 0 \\ 0 & \alpha_2 \end{pmatrix} \text{ or } \begin{pmatrix} 0 & \alpha_1 \\ \alpha_2 & 0 \end{pmatrix}$$

(when $op = \times$) or of the form

$$\begin{pmatrix} \alpha_1 & \alpha_3 \\ 0 & \alpha_2 \end{pmatrix}$$

(when $op = +$). If B is of one of those forms and $\text{lca}(x_1, x_2)$ is indeed a child of $\text{lca}(x_1, x_3)$, then the formula can be rewritten so that x_1, x_2, and x_3 all share an lca.

Suppose $op = \times$. Since B is not of the proscribed forms, either $f_B(0) \neq 0$ or $f_B(\infty) \neq \infty$. This implies that one of the two values of x_1 that block x_2 (and in the process force $\text{lca}(x_1, x_2)$'s output to 0 or ∞) does not block x_3. Similarly, one of the two values of x_2 that block x_1 does not block x_3.

Suppose $op = +$. Since B is not of the proscribed form, $f_B(\infty) \neq \infty$. This implies that the value of x_1 that blocks x_2 (forcing their lca to ∞) does not block x_3. Similarly, the value of x_2 that blocks x_1 does not block x_3. □

Thus, to build the skeleton it suffices to decide the question of whether every value of x_1 that blocks x_2 in f' will also block x_3 in f'. To find which values of x_1 block x_2 we can fix x_3 to some nonblocking value and then, as shown in the proof of Lemma 3.6, map the blocking values to the eigenvectors of a 2×2 matrix D. We proceed similarly to characterize which value(s) of x_1 block x_3. To decide if those sets of values are the same we need not calculate the eigenvectors explicitly, since two 2×2 matrices have the same eigenvectors if and only if they have the same determinant and the same trace (the sum of the two elements in the the diagonal).

3.4. Identifying functions with a single nonunary gate. To complete the process of constructing a read-once formula equivalent to f, we take the skeleton obtained from the previous steps and identify the individual gates (along with unary functions on their inputs and output). Applying Lemma 3.2, this problem reduces to identifying arithmetic read-once formulas that have a single nonunary gate (this subroutine will be invoked once for each nonunary gate in f). These formulas have the form

$$f_C(f_{A_1}(x_1)\ op\ \ldots\ op\ f_{A_n}(x_n)),$$

where $op \in \{\times, +\}$. Our skeleton from the previous sections has gates with unbounded fan-in, but without loss of generality we can split the $+$ and $\times$ gates in the skeleton so that each has fan-in two. Then our problem is to identify a two input read-once formula $f(x_1, x_2) = f_C(f_A(x_1)\ op\ f_B(x_2))$.

LEMMA 3.8. *There is an $O(1)$ time algorithm that uses membership queries to identify exactly an unknown arithmetic formula on two variables that depends on both its inputs, when the (single) nonunary operation is known. A single square root computation is also required if the nonunary operation is multiplication.*

Proof. As above, suppose $f(x_1, x_2) = f_C(f_A(x_1)\ op\ f_B(x_2))$. We first examine the case where $op = \times$. We select two values $x_2^{(1)}$ and $x_2^{(2)}$ for x_2 that do not block x_1 (we find these by trying at most four values for x_2; if the field has only three elements the single gate problem is easy since we can test all possibilities for A, B, and C). Then, using three values for x_1 and interpolating by Property 2.1 (3), we can find

$$\begin{aligned} f_{H_i}(x_1) &= f(x_1, x_2^{(i)}) \\ &= f_C(f_A(x_1) \times f_B(x_2^{(i)})) \\ &= f_{C\begin{pmatrix} b_i & 0 \\ 0 & 1 \end{pmatrix} A}(x_1), \end{aligned} \tag{6}$$

where $b_i = f_B(x_2^{(i)})$ (see Property 2.1 (7)). This gives

$$H_i = c_i C \begin{pmatrix} b_i & 0 \\ 0 & 1 \end{pmatrix} A$$

for some constant c_i (Property 2.1 (4)). Now we compute

$$D_1 = H_1^{-1} H_2 = A^{-1} \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} A, \tag{7}$$

$$D_2 = H_2 H_1^{-1} = C \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} C^{-1}, \tag{8}$$

where $\lambda_1 = c_2 b_2 / c_1 b_1$ and $\lambda_2 = c_2 / c_1$. (Matrices H_1 and H_2 are invertible since x_1 is not blocked.) We calculate the eigenvalues λ_1 and λ_2 of D_1, which are also the eigenvalues of D_2. (Here is where the square root is computed.) Note that since $b_1 \neq b_2$ the eigenvalues are distinct. Then, from equations (7) and (8) (by solving linear equations) we find A and C.

The A and C we find in this manner are not unique solutions to equations (7) and (8). But it turns out that any such A and C will suffice in the sense that for the solutions A' and C' which we find, it will be true that for some B',

$$f_C(f_A(x_1) \times f_B(x_2)) = f_{C'}(f_{A'}(x_1) \times f_{B'}(x_2)).$$

Given this claim, we may easily find the B' given A' and C'.

To prove the claim we use Property 3.9 (1) and (2) below, which imply that we shall have either $f_{A'} = \alpha f_A$ or $f_{A'} = \alpha / f_A$ (depending on which of the two eigenvalues we label λ_1). Likewise we get $f_{B'} = \beta f_B$ (in the former case) or $f_{B'} = \beta / f_B$ (in the latter). The claim follows from Property 2.4 (1).

The proof for $op = +$ proceeds similarly to the previous case. Again, we select two values $x_2^{(1)}$ and $x_2^{(2)}$ that do not block x_1, and we interpolate to find

$$\begin{aligned} f_{H_i}(x_1) &= f(x_1, x_2^{(i)}) \\ &= f_C(f_A(x_1) + f_B(x_2^{(i)})) \\ &= f_{C\begin{pmatrix} 1 & b_i \\ 0 & 1 \end{pmatrix} A}(x_1), \end{aligned} \tag{9}$$

where $b_i = f_B(x_2^{(i)})$ (see Property 2.1 (7)). This gives

$$H_i = c_i C \begin{pmatrix} 1 & b_i \\ 0 & 1 \end{pmatrix} A$$

for some constant c_i (Property 2.1 (4)). Now we compute

$$D_1 = H_1^{-1} H_2 = A^{-1} \begin{pmatrix} \lambda_1 & \lambda_2 \\ 0 & \lambda_1 \end{pmatrix} A, \tag{10}$$

$$D_2 = H_2 H_1^{-1} = C \begin{pmatrix} \lambda_1 & \lambda_2 \\ 0 & \lambda_1 \end{pmatrix} C^{-1}, \tag{11}$$

where $\lambda_1 = c_2 / c_1$ and $\lambda_2 = c_2 (b_2 - b_1) / c_1$. (Matrices H_1 and H_2 are invertible since x_1 is not blocked.) We calculate the eigenvalue λ_1 of D_1, which is also the eigenvalue of D_2. (This does not require a square root computation, since the characteristic polynomial has a zero discriminant.) Then from equations (10) and (11) (by solving linear equations) we find an A and C.

The A and C we find in this manner are not unique solutions to equations (10) and (11). But it turns out that any such A and C will suffice in the sense that for the solutions A' and C' which we find, it will be true that for some B',

$$f_C(f_A(x_1) + f_B(x_2)) = f_{C'}(f_{A'}(x_1) + f_{B'}(x_2)).$$

Given this claim, we may easily find the B' given A' and C'.

To prove the claim we use Property 3.9 (3) and (4) below, which imply that we shall have $f_{A'} = \alpha f_A + \beta$. Likewise we get $f_{B'} = \alpha f_B + \gamma$. The claim follows from Property 2.4 (3). □

We omit the straightforward proof of Property 3.9. Note that the first two subproperties follow immediately from the fact that the columns of A^{-1} and X^{-1} are eigenvectors for the two eigenvalues δ_1 and δ_2 of

$$A^{-1}\begin{pmatrix}\delta_1 & 0\\ 0 & \delta_2\end{pmatrix}A = X^{-1}\begin{pmatrix}\delta_1 & 0\\ 0 & \delta_2\end{pmatrix}X,$$

and the eigenvectors are unique up to nonzero scalar multiples (and the ordering of the eigenvalues).

PROPERTY 3.9.

(1) *If* $\delta_1 \neq \delta_2$, $\delta_1 \neq 0$, *and* $\delta_2 \neq 0$, *then*

$$A^{-1}\begin{pmatrix}\delta_1 & 0\\ 0 & \delta_2\end{pmatrix}A = X^{-1}\begin{pmatrix}\delta_1 & 0\\ 0 & \delta_2\end{pmatrix}X \text{ if and only if } X = \begin{pmatrix}\alpha_1 & 0\\ 0 & \alpha_2\end{pmatrix}A$$

for some constants α_1 *and* α_2.

(2) *If* $\delta_1 \neq \delta_2$, $\delta_1 \neq 0$, *and* $\delta_2 \neq 0$, *then*

$$A^{-1}\begin{pmatrix}\delta_1 & 0\\ 0 & \delta_2\end{pmatrix}A = X^{-1}\begin{pmatrix}\delta_2 & 0\\ 0 & \delta_1\end{pmatrix}X \text{ if and only if } X = \begin{pmatrix}0 & \alpha_1\\ \alpha_2 & 0\end{pmatrix}A$$

for some constants α_1 *and* α_2.

(3) *If* $\delta_1, \delta_2, \delta_3 \neq 0$, *then*

$$A^{-1}\begin{pmatrix}\delta_1 & \delta_2\\ 0 & \delta_1\end{pmatrix}A = X^{-1}\begin{pmatrix}\delta_1 & \delta_3\\ 0 & \delta_1\end{pmatrix}X \text{ if and only if } X = \begin{pmatrix}\alpha_1 & \alpha_3\\ 0 & \alpha_2\end{pmatrix}A$$

for some constants α_1, α_2, *and* α_3.

(4) *If* $\delta_1, \delta_2 \neq 0$, *then*

$$A^{-1}\begin{pmatrix}\delta_1 & \delta_2\\ 0 & \delta_1\end{pmatrix}A = X^{-1}\begin{pmatrix}\delta_1 & \delta_2\\ 0 & \delta_1\end{pmatrix}X \text{ if and only if } X = \begin{pmatrix}\alpha_1 & \alpha_2\\ 0 & \alpha_1\end{pmatrix}A$$

for some constants α_1 *and* α_2.

4. Finding three-justifying assignments. In this section we address the problem of how to obtain three-justifying assignments. Intuitively, the larger the field, the easier this problem, since an assignment must be justifying for any particular variable unless it sets some subtree of the formula to 0 or ∞. As the number of field elements increases, the proportion of assignments that sets some subtree to 0 or ∞ declines.

4.1. Using randomized membership queries over large fields. Lemmas 4.2 and 4.3 give randomized procedures for finding one- and three-justifying assignments, respectively. Note that the procedures draw random elements from a set of m distinct elements in $\mathcal{K}$, and the probability that the procedures succeed in finding the desired justifying assignments is dependent on m. To obtain a high probability of success we need m to be large, and hence we need $\mathcal{K}$ to contain a large number of distinct elements.

We make use of the following lemma adapted from a result of Schwartz [16].

LEMMA 4.1 [16]. *Let* $A \subseteq \mathcal{K}$ *be a finite set of field elements. If* $p(x_1, \ldots, x_n)$ *is a polynomial of degree* d *that is not identically equal to* 0, *then the total number of roots of* p *in* A^n *is at most*

$$d\,|A|^{n-1}.$$

LEMMA 4.2. *Let f be a read-once formula such that $f(x_1, x_2, \ldots, x_n)$ depends on x_1. For a sequence of t random assignments, $x_2^{(i)}, \ldots, x_n^{(i)}$, $i = 1, \ldots, t$, chosen uniformly from a set $A \subseteq \mathcal{K}$ with $|A| = m$, the projection $f(x_1, x_2^{(i)}, \ldots, x_n^{(i)})$ depends on x_1 for some i with probability at least*

$$1 - \left(\frac{2n}{m}\right)^t.$$

Proof. Since $f(x_1, \ldots, x_n)$ depends on x_1 and is an arithmetic read-once formula with respect to x_1, we have

$$f = \frac{p_{11}x_1 + p_{12}}{p_{21}x_1 + p_{22}},$$

where the p_{ij}'s do not depend on x_1 and

$$p_{11}p_{22} - p_{12}p_{21} = \det\begin{pmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{pmatrix} \not\equiv 0.$$

Note that the projection $f(x_1, x_2^{(i)}, \ldots, x_n^{(i)})$ depends on x_1 if and only if

$$(p_{11}p_{22} - p_{12}p_{21})(x_2^{(i)}, \ldots, x_n^{(i)}) \neq 0.$$

Since $\deg(p_{jk}) \leq n$, we have $\deg(p_{11}p_{22} - p_{12}p_{21}) \leq 2n$. Applying Lemma 4.1, the probability that for all $i = 1, \ldots, t$ we have $(p_{11}p_{22} - p_{12}p_{21})(x_2^{(i)}, \ldots, x_n^{(i)}) = 0$ is at most

$$\left(\frac{2n}{m}\right)^t. \qquad \square$$

LEMMA 4.3.

(1) *Let f be a read-once formula such that $f(x_1, x_2, \ldots, x_n)$ depends on x_1, x_2, and x_3. For a sequence of t random assignments, $x_4^{(i)}, \ldots, x_n^{(i)}$, $i = 1, \ldots, t$, chosen uniformly from a set $A \subseteq \mathcal{K}$ with $|A| = m$, the projection $f(x_1, x_2, x_3, x_4^{(i)}, \ldots, x_n^{(i)})$ depends on x_1, x_2, x_3 for some i with probability at least*

$$1 - \left(\frac{2n+4}{m}\right)^t.$$

(2) *Given $x_4^{(0)}, \ldots, x_n^{(0)}$, we can deterministically verify whether $f(x_1, x_2, x_3, x_4^{(0)}, \ldots, x_n^{(0)})$ depends on x_1, x_2, and x_3 using $O(1)$ membership queries.*

Proof. We prove (1) for $t = 1$. The proof for any t then follows from the independence of the random assignments. Assume (w.l.o.g.) that all nonunary gates have fan-in 2 and that $\text{lca}(x_2, x_3)$ is below $\text{lca}(x_1, x_2)$. Consider the five formulas obtained by cutting f into pieces by deleting the nodes $\text{lca}(x_1, x_2)$ and $\text{lca}(x_2, x_3)$. Each piece is a read-once formula that has exactly one input from $\{x_1, x_2, x_3\}$ or from a node that was the lca of two (or more) of those variables. When the inputs to a piece are fixed as in some random assignment to $\{x_4, \ldots, x_n\}$, a unary function is induced on the piece's one remaining input. To show that to $\{x_4, \ldots, x_n\}$ is a three-justifying assignment it is enough to show that for each of the five formulas the induced unary function is some f_A with A nonsingular. If n_1 through n_5 are the number of inputs to each of these five pieces, then Lemma 4.2 implies that the probability is at least $1 - 2n_1/m - \cdots - 2n_5/m$ that each A is nonsingular. The lemma follows since the sum of

the n_i's is at most $n + 2$ (each variable is an input to one piece and the two lca's that were deleted are also inputs).

Part 2 follows from the more general proof of Lemma 4.5 in the next section. □

Note that when the number of the elements in the field is $2n + 5$ and $t = c(2n + 5)$, Lemma 4.3 gives a technique that with probability at least $1 - e^{-c}$ finds a three-justifying assignment for any set of three variables on which the target formula depends. The randomized algorithm in this case is a Las Vegas algorithm (if we already know the set of variables on which the formula depends), since we can verify deterministically whether the projection depends on the three variables. The expected number of queries made by this algorithm is $O(n/k)$ when the field has at least $2n + 4 + k$ elements.

The algorithm for finding the variables on which the target formula depends is Monte Carlo, correctly identifying a nondependent variable with probability 1 and correctly identifying a dependent variable with probability at least $1 - (2n/m)^t$. These results together with the results in the previous section give the following theorem.

THEOREM 4.4. *There is a Las Vegas randomized polynomial time algorithm that uses membership queries to identify exactly an arbitrary n variable arithmetic read-once formula over a field, provided that field has at least $2n + 5$ elements and the variables on which the formula depends are known. The variables on which the read-once formula depends can be found in Monte Carlo randomized polynomial time provided that the field has at least $2n + 1$ elements.*

4.2. Deterministic membership query algorithms. In this section we argue (nonconstructively) that for some fields there exist nonuniform deterministic membership query algorithms. We show that there is an $O(n^5)$ time complexity nonuniform algorithm that learns arithmetic read-once formulas over fields of characteristic 0. We also show that there exists an $O(n^7 \log n)$ time complexity nonuniform algorithm that learns division-free read-once formulas over any field with at least $2n^3$ elements. Our algorithms are nonuniform in the usual sense. That is, for every n we construct a table of size poly(n) and use the table to learn formulas deterministically with at most n variables.

Let V be the set of n variables on which the target formula is defined. If $v \in \mathcal{K}^n$ is an input setting for f (and an assignment of values to V) and $\xi \in \{0, 1\}$, we denote by $v_{x_i \leftarrow \xi}$ the assignment that is equal to v in all entries except entry v_i, which is set to ξ.

Both algorithms rely on the existence of a *testing set*. This is a set of s vectors $T = \{v^{(1)}, \ldots, v^{(s)}\} \subset \mathcal{K}^n$ having the property that for any arithmetic read-once formula $f(x_1, \ldots, x_n)$ over $\mathcal{K}$ and any subset of variables $X \subseteq \{x_1, \ldots, x_n\}$, there exists some $v^{(i)} \in T$ for which $f|((V - X) \leftarrow v^{(i)})$ depends on all the variables in X if and only if f depends on all the variables in X. The following lemma shows how the existence of a small testing set implies the existence of a deterministic polynomial time identification algorithm, and in the following two subsections we show that testing sets indeed exist for the classes of formulas we consider.

LEMMA 4.5. *Suppose $T = \{v^{(1)}, \ldots, v^{(s)}\}$ is a testing set for a set F of arithmetic read-once formulas defined on n variables over the field $\mathcal{K}$. Then there is a deterministic membership query algorithm for identifying an unknown formula $f \in F$, whose running time is polynomial in n and s.*

Proof. To find the set of variables on which f depends, we check each x_i and $v^{(j)} \in T$ to see whether $f(v^{(j)}_{x_i \leftarrow 0}) = f(v^{(j)}_{x_i \leftarrow 1})$. If f depends on x_i, then for some $v^{(j)}$, $f|((V - \{x_i\}) \leftarrow v^{(j)})$ depends on x_i, and hence by Property 2.1 (1) these two values of f will differ (and not be ERROR). Then, to apply our previous techniques (Lemma 3.1) we need only show how to decide which $v^{(j)} \in T$ is a two- or three-justifying assignment for each subset X of two or three variables on which f depends. Let $k = |X|$ (≤ 3). Let $A \subset \mathcal{K}$ be a set of $m > 18$ field elements (if $\mathcal{K}$ contains fewer elements we can check whether the assignment is justifying by trying all

possible settings of the k variables in constant time). By Lemma 4.2, if $f|((V - X) \leftarrow v)$ depends on each $x_i \in X$ then it does not depend on x_i for at most a fraction $2k/m$ of the m^k input vectors from A^k to which we might set the variables in X. There can be at most $3(2km^{k-1})$ settings for X in which f fails to depend on any of the at most three variables in X, and since $m > 6k$ this quantity is less than m^k, implying that some input vector from A^k will give a justifying assignment. Note that m^k is $O(1)$. Thus finding justifying assignments from the testing set requires $O(sn^3)$ time and membership queries. □

Note that the previous section shows that once you fix a formula, the probability is very high that a small set of random assignments will have all the necessary justifying assignments. If the number of possible formulas were small (or even exponential in n) we could then show that the probability a random set of assignments fails to have the necessary justifying assignments for *any* f less than 1, implying that a testing set exists. Unfortunately, the number of formulas is very large (or infinite) depending on the field, so this simplistic argument will not work. The following two sections more carefully demonstrate the existence of testing sets using more elaborate probabilistic (and hence nonconstructive) methods.

4.2.1. A testing set for fields of characteristic 0. For fields of characteristic 0 our existence proof for a testing set is built upon the following lemma of Heintz and Schnorr.

LEMMA 4.6 [13]. *Let $\mathcal{K}$ be a field of characteristic 0. Let $P_{d,\mu}$ be the set of all polynomials over $\mathcal{K}$ on variables $x_1, \ldots, x_n$ that have degree at most d and that can be computed with a circuit using at most μ nonscalar multiplications/divisions. Then for $u = 2\mu(d+1)$ and $s = 6(\mu+1)(\mu+2)$ there exist $v^{(1)}, \ldots, v^{(s)} \in \{1, 2, \ldots, u\}^n$ such that for any $p \in P_{d,\mu}$,*

$$p(v^{(1)}) = \cdots = p(v^{(s)}) = 0 \text{ if and only if } p \equiv 0.$$

Based on the Heintz–Schnorr lemma we prove the following result.

LEMMA 4.7. *Let $\mathcal{K}$ be a field of characteristic 0. Let F be the set of all arithmetic read-once formulas. There exist $s = 6(7n-7)(7n-6)$ vectors $T = \{v^{(1)}, \ldots, v^{(s)}\} \subseteq \{0, 1, \ldots, 2(7n-8)(4n^2-5n+1)\}^n$ with the following property: for every $f \in F$ there is a vector $v^{(i_f)} \in T$, where for each subformula g of f that is the input to a $+$ or $\times$ gate,*

$$g(v^{(i_f)}) \notin \{0, \infty, \text{ERROR}\}.$$

Proof. Let f be any arithmetic read-once formula. The tree corresponding to f can be regarded as a circuit C that computes f. We now show how to change this circuit to a new circuit C' satisfying the following condition: the output of C' for some vector input v is 0 if and only if $g(v) \in \{0, \infty, \text{ERROR}\}$ for some subformula g of f. We construct C' from C as follows: First, each node α in C that computes a rational function f_α will map to two nodes α_1 and α_2 in C'. The nodes α_1 and α_2 will compute polynomials f_{α_1} and f_{α_2}, respectively, where $f_\alpha = f_{\alpha_1}/f_{\alpha_2}$.

(1) If node α in C is labeled with a unary function $f_{\binom{a\ b}{c\ d}}$ and is the parent of a node β in C, then we define two nodes α_1 and α_2 in C' that compute

$$f_{\alpha_1} = af_{\beta_1} + bf_{\beta_2} \text{ and } f_{\alpha_2} = cf_{\beta_1} + df_{\beta_2}.$$

(2) If node α in C is labeled with multiplication $\times$ and is the parent of nodes β and γ, then we define two nodes α_1 and α_2 in C' that compute

$$f_{\alpha_1} = f_{\beta_1} f_{\gamma_1} \text{ and } f_{\alpha_2} = f_{\beta_2} f_{\gamma_2}.$$

(3) If node α in C is labeled with addition $+$ and is the parent of nodes β and γ, then we define two nodes α_1 and α_2 in C' that compute

$$f_{\alpha_1} = f_{\beta_1} f_{\gamma_2} + f_{\gamma_1} f_{\beta_2} \text{ and } f_{\alpha_2} = f_{\beta_2} f_{\gamma_2}.$$

It is easy to see that the root r in C corresponds to two nodes r_1 and r_2 in C', where $f = f_r = f_{r_1}/f_{r_2}$.

In f there are $2n-2$ subformulas that are inputs to $+$ or $\times$ gates. Each such g can have $g(v) \in \{0, \infty, \text{ERROR}\}$ only if one of the two corresponding nodes in C' is 0. We add to the root of C' $4n-5$ multiplication gates that multiply all these nodes, and it is now obvious that the output of C' is 0 for some vector input v if and only if some $g(v) \in \{0, \infty, \text{ERROR}\}$. The nonscalar multiplicative complexity of the circuit C' is at most $(4n-5)+3(n-1) = 7n-8$, and the degree of the polynomial that is computed in C' is at most $n(4n-5)$. Therefore, by Lemma 4.6 there exist $s = 6(7n-7)(7n-6)$ vectors $T = \{v^{(1)}, \ldots, v^{(s)}\} \subseteq \{0, 1, \ldots, 2(7n-8)(4n^2-5n+1)\}^n$ that satisfy the condition of the lemma. □

From this we can prove our desired result.

THEOREM 4.8. *There is a deterministic polynomial time algorithm that uses membership queries to identify exactly an arbitrary n variable arithmetic read-once formula over a field of characteristic* 0.

Proof. Lemma 4.7 states that there exists a testing set of size $O(n^2)$. The result follows from Lemma 4.5. □

4.2.2. A testing set for division-free formulas over large fields. In the division-free case we may consider the basis functions to be $\times$, $+$, and $(a|b)$, where $(a|b)(x) = ax + b$. We assume that the $\times$ and $+$ gates have fan-in two.

We define a tree $\Gamma(f)$ obtained from f by removing all nodes labeled with $(a|b)$ by replacing each input x_i by ω_i and changing the labels $\times$ to $+$ and the labels $+$ to $\overline{\max}$. The computation in the formula $\Gamma(f)$ is defined by the following rules:

(1) The inputs $(\omega_1, \ldots, \omega_n)$ must be positive integers.

(2) A node labeled with $+$ computes the sum of the two results in its children.

(3) A node labeled with $\overline{\max}(x, y)$ computes the maximum of x and y if $x \neq y$ and gives the result ND (not defined) if $x = y$.

(4) ND$+x$ =ND and $\overline{\max}$(ND, y) =ND for any $x, y \in \{\text{ND}, 1, 2, \ldots\}$.

The function computed by $\Gamma(f)$ is denoted by f_Γ. The connection between f_Γ and f is described in the following lemma. We omit the simple proof.

LEMMA 4.9. *Let $\omega_1, \ldots, \omega_n$ be positive integers. If $f_\Gamma(\omega_1, \ldots, \omega_n) \neq$ND then*

$$\deg f(x^{\omega_1}, \ldots, x^{\omega_n}) = f_\Gamma(\omega_1, \ldots, \omega_n).$$

We add ND to the computation of the tree because when we have subtraction of two polynomials of the same degree, we cannot know the degree of the result. An immediate consequence of Lemma 4.9 is the following lemma.

LEMMA 4.10. *If $f_\Gamma(\omega_1, \ldots, \omega_n) \neq$ND then $f(x^{\omega_1}, \ldots, x^{\omega_n}) \not\equiv 0$.*

The following lemma shows that in fact most input settings (where the ω_i's have values between 1 and a sufficiently large c) do not cause f_Γ to output ND. From this we shall be able to show that there is a polynomial size set of input settings such that no f_Γ outputs ND on all of them, and from that we shall be able to prove the existence of a testing set for division-free read-once formulas (provided the field has sufficiently many elements).

LEMMA 4.11. *Let $f(x_1, \ldots, x_n)$ be any arithmetic read-once formula not equivalent to* 0. *For random integers $\omega_i \in \{1, 2, \ldots, c\}$ we have*

$$\text{Prob}(f_\Gamma(\omega_1, \ldots, \omega_n) = \text{ND}) \leq \frac{n^2}{c}.$$

Proof. We shall prove the claim that for any integer k and any gate in f_Γ, the probability that the gate outputs k is at most n/c. If this is true it follows that the probability that two

inputs to a $\overline{\max}$ gate are equal is at most n/c, from which the lemma easily follows (since there are at most n such gates).

We prove the claim by induction on the number of variables n appearing in f_Γ. If $n = 1$ then f_Γ is just a single variable, and the result is trivial.

If the root of f_Γ is $+$ (i.e., $f_\Gamma = g_1 + g_2$), and if K is the (finite) possible set of integer outputs for g_1 when its variables are chosen from $\{1, \ldots, c\}$, then

$$\begin{aligned}\text{Prob}(f_\Gamma(\omega) = k) &= \sum_{k_1 \in K} \text{Prob}(g_1(\omega) = k_1)\text{Prob}(g_2(\omega) = k - k_1) \\ &\leq \sum_{k_1 \in K} \text{Prob}(g_1(\omega) = k_1)\frac{n}{c} \\ &\leq \frac{n}{c}\end{aligned}$$

(applying the inductive hypothesis on g_1, which contains fewer than n variables).

If $f_\Gamma = \overline{\max}(g_1, g_2)$, and g_1 and g_2 contain n_1 and n_2 variables, respectively, $(n_1 + n_2 = n)$, then

$$\begin{aligned}\text{Prob}(\overline{\max}(g_1, g_2) = k) &\leq \text{Prob}(g_1 = k) + \text{Prob}(g_2 = k) \\ &\leq \frac{n_1}{k} + \frac{n_2}{k},\end{aligned}$$

and the claim, and hence the lemma, follows. □

The following upper bound for the number of read-once formulas over $+$ and $\overline{\max}$ with n variables follows from simple induction.

LEMMA 4.12. *The number of read-once formulas over the operations $+$ and $\overline{\max}$ is less than n^n.*

Proof. We show that there are at most $n!$ such formulas. This is true for $n = 2$, where the only possibilities are $x_1 + x_2$ and $\overline{\max}(x_1, x_2)$. For a formula over $n + 1$ variables the inductive hypothesis applies to the two subformulas of the root (over k and $n - k$ variables, respectively). We bound the number of formulas over $n + 1$ variables as follows (the 2 comes from the choice of $+$ or $\overline{\max}$ for the root, and the $\frac{1}{2}$ comes from the fact that each formula is counted twice, since the left and right subformulas may be exchanged):

$$2 \cdot \frac{1}{2} \cdot \sum_{k=1}^{n-1} k!(n-k)! \;\leq\; (n-1)n! \;\leq (n+1)!\,. \qquad \square$$

Now we can prove the key result of this section that makes the transition from a randomized to a deterministic algorithm.

LEMMA 4.13. *There exists a set $\{\omega^{(1)}, \ldots, \omega^{(m)}\} \subseteq \{1, 2, \ldots, n^{2+\epsilon}\}^n$, where $m = \lceil \frac{n}{\epsilon} \rceil$ such that for any arithmetic read-once formula f not equivalent to 0, one of $f_\Gamma(\omega^{(1)}), \ldots, f_\Gamma(\omega^{(m)})$ is not* ND.

Proof. Choose $\omega^{(1)}, \ldots, \omega^{(m)}$ randomly. By Lemma 4.11 we have

$$\text{Prob}(f_\Gamma(\omega^{(i)}) = \text{ND}) \leq \frac{n^2}{c} = n^{-\epsilon}.$$

Therefore

$$\text{Prob}((\forall i \leq m) f_\Gamma(\omega^{(i)}) = \text{ND}) \leq (n^{-\epsilon})^m \leq n^{-n},$$

and by Lemma 4.12

$$\text{Prob}((\exists f_\Gamma)(\forall i \leq m) f_\Gamma(\omega^{(i)}) = \text{ND}) < n^n n^{-n} = 1.$$

Therefore, there exist $\omega^{(1)}, \ldots, \omega^{(m)}$ such that for any f_Γ, one of $f_\Gamma(\omega^{(1)}), \ldots, f_\Gamma(\omega^{(m)})$ is not ND. □

LEMMA 4.14. *Let $\kappa_1, \ldots, \kappa_{2n^3+1}$ be any distinct elements in K. There exists*

$$T = \{v^{(1)}, \ldots, v^{(s)}\} \subseteq \left(\bigcup_{i=1}^{2n^2} \{\kappa_1^i, \ldots, \kappa_{(2n^3+1)}^i\}\right)^n,$$

where $s \leq 3n^4 \log n$, such that for any arithmetic read-once formula f we have

$$f(v^{(1)}) = \cdots = f(v^{(s)}) = 0 \quad \textit{if and only if} \quad f \equiv 0.$$

Proof. For $\omega = (\omega_1, \ldots, \omega_n)$ we will denote $(x^{\omega_1}, \ldots, x^{\omega_n})$ by x^ω.
Let $\omega^{(1)}, \ldots, \omega^{(m)}$ be the vectors in Lemma 4.13 with $\epsilon = \frac{1}{\log n}$. We define

$$T = \{\gamma^{\omega^{(i)}} | i = 1, \ldots, m \text{ and } \gamma = \kappa_1, \kappa_2, \ldots, \kappa_{2n^3+1}\}.$$

The entries of $\gamma^{\omega^{(i)}}$ are from $\bigcup_{i=1}^{2n^2} \{\kappa_1^i, \ldots, \kappa_{(2n^3+1)}^i\}$ (using $2n^2$ as an upper bound for $n^{2+\epsilon}$) and

$$|T| = m(2n^3 + 1) \leq 3n^4 \log n.$$

Now, if $f \equiv 0$ then $f(v) = 0$ for all $v \in T$. If $f \not\equiv 0$ then by Lemma 4.13 there exist $\omega^{(i)}$ such that $f_\Gamma(\omega^{(i)}) \neq$ND, which by Lemma 4.10 implies that $f(x^{\omega^{(i)}}) \not\equiv 0$. Note that $f(x^{\omega^{(i)}})$ is a polynomial over one variable with

$$\deg f(x^{\omega^{(i)}}) \leq \sum_{j=1}^{n} \omega_j^{(i)} \leq n(2n^2) = 2n^3.$$

Thus $f(x^{\omega^{(i)}})$ can have at most $2n^3$ roots, and hence for some $\gamma_0^{\omega^{(i)}}$, $\gamma_0 \in \{\kappa_1, \ldots, \kappa_{2n^3+1}\}$ we must have $f(\gamma_0^{\omega^{(i)}}) \neq 0$. □

To prove the main theorem for this section we show that T will indeed be a testing set.

THEOREM 4.15. *There is a nonuniform deterministic polynomial time algorithm that uses membership queries to identify exactly an arbitrary n variable division-free arithmetic read-once formula over a field with at least $2n^3 + 1$ elements.*

Proof. Using Lemma 4.5 we need to demonstrate the existence of a testing set. We claim that the set T described by Lemma 4.14 is a testing set.

Let f be any division-free arithmetic read-once formula. Let α be any node in f, f_α be the formula computed at node α, and X_α be the variables in f_α. We will say that f_α is *maximal X-independent* if the node α is labeled with $\times$, $X_\alpha \cap X = \emptyset$, and if for the parent $p(\alpha)$ of α, $X_{p(\alpha)} \cap X \neq \emptyset$. It is obvious that for any two maximal X-independent formulas f_α and f_β, X_α and X_β are either equal of disjoint. Let $f_{\alpha_1}, \ldots, f_{\alpha_t}$ be all the maximal X-independent formulas in f. By the previous properties we have that

$$h = f_{\alpha_1} \ldots f_{\alpha_t}$$

is a division-free arithmetic read-once formula. By Lemma 4.14, for some $v^{(i)}$ we have $h(v^{(i)}) \neq 0$, which is equivalent to $f_{\alpha_j}(v^{(i)}) \neq 0$ for $j = 1, \ldots, t$. Now it can be easily shown that if f depends on all the variables of X if and only if all the maximal X-independent formulas are not zero for $v^{(i)}$, $f|((V - X) \leftarrow v^{(i)})$ therefore depends on all the variables of X. □

4.3. Using equivalence queries over arbitrary fields. An alternate technique for generating justifying assignments requires equivalence queries but works for small as well as large fields. The basic approach is to use the core algorithm to learn a projection of the target formula, where the variables for which we do not yet have justifying assignments are fixed. We then make an equivalence query, and the counterexample is used to find justifying assignments for a new variable.

THEOREM 4.16. *There is a deterministic polynomial time algorithm that uses membership and equivalence queries to identify exactly an arbitrary n variable arithmetic read-once formula over a field, provided that field has at least* 3 *elements.*

Proof. The algorithm starts with an equivalence query on some arbitrary constant hypothesis. Assuming the answer is not "yes," the result is a counterexample $a = (a_1, \ldots, a_n)$ for which $f(a) \neq \text{ERROR}$ (if f is not constant it must take on at least two different non-ERROR values). At this point we describe all variables as "static," meaning we have no justifying assignment for any of them.

The algorithm works in phases. During a phase, each variable is categorized either as "active" or "static." The algorithm has a justifying assignment for each active variable. In addition, it has two- and three-justifying assignments for each pair and triple of active variables. Furthermore, all these justifying assignments assign the same values to each static variable.

At the start of a phase, the algorithm tries to increase the number of active variables by repeating the following procedure for each currently active variable, which we call the *activation procedure.* Assume w.l.o.g. that the static variables are $x_{m+1}, \ldots, x_n$ and that they are assigned values $a_{m+1}, \ldots, a_n$. (For the initial case where no variables are active, the activation procedure is simply to search for a justifying assignment for some variable x_i by setting a_i to two different values and checking whether the value of f changes.)

1. Pick some currently active variable x_j with justifying assignment $a_1, \ldots, a_m, a_{m+1}, \ldots, a_n$.
2. Check whether the projection

$$f(a_1, \ldots, a_{j-1}, x_j, a_{j+1}, \ldots, a_{i-1}, x_i, a_{i+1}, \ldots, a_n)$$

depends on both its inputs x_i and x_j.

3. If $m > 1$, check whether for each other current active variable x_k, we can obtain a two-justifying assignment for x_k and x_i by taking the two-justifying assignment for x_k and x_j, unsetting x_i, and setting x_j to some value that does not block x_i (obtained by trying three arbitrary values).
4. If $m > 2$, similarly check whether each three-justifying assignment for x_j and a pair of active variables can be converted to a three-justifying assignment for that pair of variables and x_i. If there are only two current active variables x_j and x_k, then check whether taking their two-justifying assignment and unsetting x_i gives a three-justifying assignment for x_i, x_j, and x_k.
5. If all the above conditions are true for x_i, then make x_i active. Save the two and three-justifying discovered above. Find and save a justifying assignment for x_i by taking one of the two-justifying assignments for x_i and some other active variable x_j, and setting x_j so it doesn't block x_i (obtained by trying three arbitrary values).

When the activation procedure fails to find any more new active variables, the algorithm learns the projection $f(x_1, \ldots, x_{m+1}, a_{m+1}, \ldots, a_n)$, in which the static variables $x_{m+1}, \ldots, x_n$ are fixed to the values $a_{m+1}, \ldots, a_n$. The algorithm learns this projection by executing our core algorithm, using the justifying assignments associated with the active variables (note that these assignments are justifying both for f and for the projection). The algorithm then performs an

equivalence query "$h \equiv f$?" using the learned projection as the hypothesis h. If the answer is "yes," the algorithm is done (this happens precisely when all relevant variables of f are active, because then and only then is the projection equal to f). Otherwise, the algorithm receives a counterexample $v = (b_1, \ldots, b_n)$ for which $h(v) \neq f(v)$ (recall that $f(v) \neq$ ERROR if possible, and we prove below that such a nonerror counterexample is always available). We shall show how to use this counterexample to increase the number of active variables.

The algorithm then processes the counterexample as follows in order to find more new active variables: It chooses an arbitrary static variable x_i such that $a_i \neq b_i$ (one must exist since h is correct on the projection where $x_{m+1}, \ldots, x_n$ are set to $a_{m+1}, \ldots, a_n$). It tests three possible field values to a_i' to assign for x_i in search of one for which

$$f(x_1, \ldots, x_m, a_{m+1}, \ldots, a_n) \equiv f(x_1, \ldots, x_m, a_{m+1}, \ldots, a_{i-1}, a_i', a_{i+1}, \ldots, a_n)$$

and

$$f(b_1, \ldots, b_{m-1}, a_{m+1}, \ldots, a_{i-1}, a_i', a_{i+1}, \ldots, a_n) \neq f(b_1, \ldots, b_{i-1}, a_i', b_{i+1}, \ldots, b_n),$$

and neither of the quantities in the second condition is ERROR. To check the first condition it would suffice to verify that none of the justifying assignments or membership queries made while running the core algorithm to find h are affected by the change (though since the projection has special properties, the condition can in fact be checked with one membership query, as we describe below). The second condition is easily checked with a membership query, as is the non-ERROR condition.

If such an a_i' is found, the algorithm updates a_i and b_i to a_i'. It also updates a_i to a_i' in the justifying assignments associated with the active variables. We argue below that such an a_i' must exist (given that the activation procedure failed on the current set of active variables and justifying assignments).

The algorithm then tries to use the activation procedure (with the new value a_i') to find a new active variable. If the procedure fails, the algorithm repeats the above process, making the a_i's and b_i's agree on another variable and again attempting the activation procedure. Since the changes still leave the modified v a counterexample to $h \equiv f$ (by the second condition, whose left-hand side is $h(v)$ and whose right-hand side is $f(v)$), the algorithm must eventually find some new active variable (given the claim that it can otherwise always find a suitable a_i'). Once it has found a new active variable it begins a new phase with the expanded set of active variables.

Because the set of active variables grows at each phase, the algorithm eventually makes all relevant variables active and learns f.

This completes the description of the algorithm. We now justify the claim that in the processing of the counterexample, if the activation procedure failed with the current set of active variables and justifying assignments, then a_i' exists. Consider the processing of the counterexample b. A static variable x_i is chosen such that $a_i \neq b_i$. The procedure for finding a new active variable failed, so, in particular, it failed to make x_i an active variable. We consider the conditions that caused this to occur.

Let $\hat{f}$ be the maximal subformula of f that contains x_i, but no active variables. Let $f_A(x_i)$ be the unary function induced on $\hat{f}$ when its remaining (static) inputs are set as in $a_{m+1}, \ldots, a_n$. We shall show that because the activation procedure didn't make x_i active, $\det(A) = 0$. Assume not. Then $f_A(x_i)$ depends on x_i. Let x_j be the active variable with an lca as deep as possible with x_i (i.e., lca(x_i,x_j) is the gate in f for which $\hat{f}$ is an input). Let x_j's justifying assignment be $a_1, \ldots, a_n$. The projection considered in step 2 of the activation procedure is

$$f(a_1, \ldots, a_{j-1}, x_j, a_{j+1}, \ldots, a_{i-1}, x_i, a_{i+1}, \ldots, a_n) = f_C(f_A(x_i) \; op \; f_B(x_j)),$$

where $\det(B), \det(C) \neq 0$ (since this projection depends on x_j). The projection hence depends on both x_i and x_j. Furthermore, any two- or three-justifying assignment for a set of active variables that includes x_j must induce a projection that depends on x_i when that variable is unassigned. Finally, if x_j and some other x_k are the only active variables, then the two-justifying assignment for x_j, x_k sets the other variables precisely as in this projection (because all other variables are static); thus, unsetting x_i in the two-justifying assignment for x_j and x_k gives a three-justifying assignment for x_i, x_j, and x_k. It follows that the procedure could have made x_i active by the above choice of x_j, which is a contradiction. Therefore, $\det(A) = 0$.

It follows that setting a_i' to any value (except possibly one which yields ERROR) leaves f_A, and hence $f(x_1, \ldots, x_m, a_{m+1}, \ldots, a_n)$, unchanged. (Note that the algorithm can detect the one bad value, since this forces the output of the projection of f to ERROR on any setting of the active variables. Thus, testing the first condition for a_i' in fact takes only one membership query.) Furthermore, if the induced projection on f remains unchanged, all the justifying assignments for the active variables remain justifying. Now consider the projection

$$f_B(x_i) = f(b_1, \ldots, b_{i-1}, x_i, b_{i+1}, \ldots, b_n).$$

If $\det(B) = 0$, then changing x_i to any but at most one value does not change f_B and hence preserves $f(v) \neq h(v)$ and $f(v) \neq$ ERROR. If $\det(B) \neq 0$, then there can only be one "bad" value of x_i that sets $f_B(x_i) = h(v)$ because $h(v)$ does not depend on x_i (the algorithm can detect this one bad value using a membership query). Moreover, there are no values of x_i that set $f_B(x_i)$ to ERROR. Hence in either case, there is only one value for x_i such that v does not remain a non-ERROR counterexample to $h \equiv f$.

Thus, as claimed, out of three possible field elements to set a_i and b_i, at least one of these is not "bad" and preserves the two conditions. We can easily check whether this is the case.

To conclude the proof, we observe that there will indeed always be a non-ERROR counterexample available to return if $h \not\equiv f$ (recall that we shall get such a counterexample whenever possible). This follows from the fact that a justifying assignment exists for every static variable x_i appearing in the formula, which means that there exist inputs differing only on x_i's value that induce all possible values on f's output. The hypothesis h does not include x_i, so it cannot be correct on more than one of these inputs. □

The above algorithm requires at most n equivalence queries, because each counterexample is processed to produce at least one new active variable. The main time and membership query requirements come from the (at most) n applications of the core algorithm and the fact that from each counterexample there may be up to n attempts of the activation algorithm, each of which requires $O(n^2)$ time and membership queries for every pair of a static x_i and active x_j.

5. Lower bounds.

5.1. Lower bounds for small fields. In this section we show that the identification results we achieve using (nonstandard) equivalence queries are not achievable (at least over small fields) in the standard equivalence query model. Note that our results leave a gap of size $O(\log n)$ between the size of the largest field that, provably, requires (modified) equivalence queries and the size of the smallest field for which membership queries alone are shown to be adequate.

THEOREM 5.1. *There is no polynomial time algorithm that uses only membership and standard equivalence queries to identify exactly arithmetic read-once formulas over n variables on fields that have fewer than $o(n/\log n)$ elements.*

Proof. We consider the case where the target formula f over the field $\mathcal{K}$ is equivalent to a formula of the following form, where the variables are $V = \{x_1, \ldots, x_m, y_1, \ldots y_m\}$ $(n = 2m)$

and $a_i, b_i \in \mathcal{K}$:

$$(x_1 - a_1) \times \cdots \times (x_m - a_m) \times \frac{1}{(y_1 - b_1)} \times \cdots \times \frac{1}{(y_m - b_m)}.$$

Note that such a formula can be rewritten to contain one $\times$ gate and n unary gates.

Consider an algorithm A that uses queries to identify f. If $\mathcal{K}$ is finite and $c = |\mathcal{K}|$ is sufficiently small, we shall show with an adversary argument that a polynomial number of queries are not enough to identify f. This is true because there will not be enough information to determine the a_i's and b_i's uniquely.

Consider a membership query of A that is answered with ERROR, or an equivalence query on a rational function that is answered with "no" and a counterexample for which the value of f is ERROR (note that the hypothesis ERROR is not allowed for an equivalence query in this model).

Each such query (membership or equivalence) gives us one new example on which the value of f is ERROR. Each such example eliminates from consideration only those target functions f for which one of the following two conditions is true on the input:

(a) None of the $(x_i - a_i)$'s are 0.

(b) None of the $(y_i - b_i)$'s are 0.

Thus an ERROR example eliminates $(c-1)^m$ choices for the a_i's and $(c-1)^m$ choices for the b_i's. Hence the number of possible target formulas eliminated is less than $2(c-1)^m c^m$ (it is actually that quantity minus $(c-1)^{2m}$).

There are c^n choices for f, no two of which are equivalent. Therefore, by repeatedly giving answers of the type described before, the adversary can force the algorithm to make a number of queries that exceeds

$$\frac{c^n}{2(c-1)^{n/2}c^{n/2}} = \frac{1}{2}\left(\frac{c}{c-1}\right)^{n/2}.$$

If c is asymptotically less than any positive constant times $n/\log(n)$, this grows superpolynomially in n. This means that if the size of the field is not within a log factor of the number of variables, membership and standard equivalence queries do not suffice for polynomial time identification. □

5.2. A tight bound on the number of square root operations. In this section we show that any algorithm that exactly identifies arithmetic read-once formulas on n variables over a field $\mathcal{K}$ can be modified to an algorithm that finds the square root of $n-1$ elements in the field $\mathcal{K}$. This reduction shows that any algorithm for identifying read-once formulas should (in the worst case) compute the square root of $n-1$ elements in the field. Since our algorithm in this paper needs to compute only $n-1$ square roots, this lower bound is tight.

THEOREM 5.2. *Any algorithm that exactly identifies arithmetic read-once formulas over a field $\mathcal{K}$ must (in the worst case) compute the square root of exactly $n-1$ elements of the field $\mathcal{K}$.*

Proof. Let $\mathcal{K}$ be a field of characteristic other than 2. Consider the formula

$$f(x_1, x_2) = \frac{b^2(x_1 - x_2)}{x_1 x_2 - b^2} = f_{\left(\begin{smallmatrix} b & -b \\ 1 & 1 \end{smallmatrix}\right)}\left(f_{\left(\begin{smallmatrix} 1 & -b \\ 1 & b \end{smallmatrix}\right)}(x_1) \times f_{\left(\begin{smallmatrix} 1 & b \\ 1 & -b \end{smallmatrix}\right)}(x_2)\right).$$

When we identify the arithmetic read-once formula $f(x_1, x_2)$, we find three matrices A', B', and C', where

$$f_{C'}(f_{A'}(x_1) \times f_{B'}(x_2)) = f_{\left(\begin{smallmatrix} b & -b \\ 1 & 1 \end{smallmatrix}\right)}\left(f_{\left(\begin{smallmatrix} 1 & -b \\ 1 & b \end{smallmatrix}\right)}(x_1) \times f_{\left(\begin{smallmatrix} 1 & b \\ 1 & -b \end{smallmatrix}\right)}(x_2)\right).$$

By Property 2.4.1, we have either

$$A' = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} \alpha & -\alpha b \\ \beta & \beta b \end{pmatrix} \quad \text{or} \quad A' = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} \alpha & \alpha b \\ \beta & -\beta b \end{pmatrix}$$

for some constants α and β. In both cases a_{12}/a_{11} and a_{22}/a_{21} will be the roots of b^2. This gives a reduction from identifying a read-once formula with two variables to finding the square root of one element in the field.

We now define the following read-once formula with n variables $x_1, \ldots, x_n$:

$$f_2(x_1, x_2) = \frac{b_1^2(x_1 - x_2)}{x_1 x_2 - b_1^2}$$

and

$$f_{i+1}(x_1, \ldots, x_{i+1}) = \frac{b_i^2(x_{i+1} - f_i(x_1, \ldots, x_i))}{f_i(x_1, \ldots, x_i)x_{i+1} - b_i^2}.$$

We claim that an algorithm that identifies the formula $f_n(x_1, \ldots, x_n)$ finds the square root of $b_1^2, b_2^2, \ldots, b_n^2$. It can be easily shown that this formula is read-once and the unary operation attached to each variable x_i is

$$A' = \begin{pmatrix} a_{11}^{(i)} & a_{12}^{(i)} \\ a_{21}^{(i)} & a_{22}^{(i)} \end{pmatrix} = \begin{pmatrix} \alpha_i & -\alpha_i b_i \\ \beta_i & \beta_i b_i \end{pmatrix} \quad \text{or} \quad A' = \begin{pmatrix} a_{11}^{(i)} & a_{12}^{(i)} \\ a_{21}^{(i)} & a_{22}^{(i)} \end{pmatrix} = \begin{pmatrix} \alpha_i & \alpha_i b_i \\ \beta_i & -\beta_i b_i \end{pmatrix}$$

for some constants α_i and β_i. Therefore, identifying the formula will also give the square roots of the b_i^2's ($b_i = \pm a_{12}^{(i)}/a_{11}^{(i)}$). This completes the proof for fields of characteristic other than 2.

For fields of characteristic 2, we have instead

$$f(x_1, x_2) = \frac{b^2(x_1 + x_2)}{x_1 x_2 + b^2} = f_{\left(\begin{smallmatrix} b & 0 \\ 1 & 1 \end{smallmatrix}\right)}\left(f_{\left(\begin{smallmatrix} 1 & 0 \\ 1 & b \end{smallmatrix}\right)}(x_1) + f_{\left(\begin{smallmatrix} 1 & 0 \\ 1 & b \end{smallmatrix}\right)}(x_2)\right).$$

When we identify this formula we get $f(x_1, x_2) = f_{C'}(f_{A'}(x_1) + f_{B'}(x_2))$, for which (by Property 2.4.(2))

$$A' = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} \alpha_1 + \alpha_2 & \alpha_2 b \\ \alpha_3 & \alpha_3 b \end{pmatrix}$$

for some constants α_1, α_2, and α_3. Then a_{22}/a_{21} is the root of b^2. As before we identify the formula f_n, where now

$$f_2(x_1, x_2) = \frac{b_1^2(x_1 + x_2)}{x_1 x_2 + b_1^2}$$

and

$$f_{i+1}(x_1, \ldots, x_{i+1}) = \frac{b_i^2(x_{i+1} + f_i(x_1, \ldots, x_i))}{f_i(x_1, \ldots, x_i)x_{i+1} + b_i^2}$$

to reveal the roots of b_i^2 for $i = 1, \ldots, n-1$. □

TABLE 5.1
Summary of results.

Type of result	Randomized/ deterministic	Field size	# Queries memb.	# Queries equiv.	Complexity time	Complexity sq. roots
Upper bound	rand.	$\geq 2n+5$	$O(n^3)$		$O(n^4)$	$n-1$
Upper bound (division-free)	deter.	$\geq 2n^3+1$	$O(n^7 \log n)$		$O(n^8 \log n)$	$n-1$
Upper bound	deter.	char. 0	$O(n^5)$		$O(n^6)$	$n-1$
Upper bound	deter.	any	$O(n^6)$	n	$O(n^6)$	$n-1$
Lower bound	either	finite ($\lvert\mathcal{K}\rvert$)	a total of $\frac{1}{2}\left(\frac{\lvert\mathcal{K}\rvert}{\lvert\mathcal{K}\rvert-1}\right)^{n/2}$			
Lower bound	either	any				$n-1$

6. Summary of results. Table 5.1 summarizes the results of this paper. For the lower bound the equivalence queries are standard and for the upper bound they are in our modified form. We have shown the equivalence query upper bound for fields of three or more elements, but if we disallow division for two element fields it holds in that case as well [11], [5].

There is no consensus in the literature on how much time should be charged for the setup of a membership query. If those costs are considered constant, we can save a factor of n from the running time bound for the nonuniform deterministic algorithms.

7. Appendix. *Proof of Property* 1. Note that

$$f_A(x) = \begin{cases} \frac{a}{c} + \frac{bc-ad}{c(cx+d)} & c \neq 0, \\ \frac{ax+b}{d} & c = 0. \end{cases}$$

If $c \neq 0$ we have $f_A(x) = \frac{a}{c} - \frac{\det(A)}{c(cx+d)}$, and if $c = 0$ we have $f_A(x) = (\det(A)x + bd)/d^2$. From these it easily follows that if $\det(A) = 0$ then f_A is a constant function, except possibly on one input value where it is ERROR. And if $\det(A) \neq 0$ then f_A is a one to one mapping (or bijection) from $\mathcal{K} \cup \{\infty\}$ onto itself (if $c = 0$ and $\det(A) \neq 0$ then $d \neq 0$). This proves Property 2.1 (1).

Property 2.1 (2) follows from the fact that

$$f_{\lambda A}(x) = \frac{(\lambda a)x + (\lambda b)}{(\lambda c)x + (\lambda d)} = \frac{ax+b}{cx+d} = f_A(x).$$

Property 2.1. (3(a)) is obvious. To show Property 2.1 (3(b)) we prove that the equations $f_A(x_1) = y_1$, $f_A(x_2) = y_2$, $f_A(x_3) = y_3$ have a unique solution. We have three simultaneous equations

$$\begin{aligned} x_1a + b &= (y_1x_1)c + y_1d, \\ x_2a + b &= (y_2x_2)c + y_2d, \\ x_3a + b &= (y_3x_3)c + y_3d, \end{aligned}$$

or

$$\begin{aligned} x_1a - y_1d + b &= (y_1x_1)c, \\ x_2a - y_2d + b &= (y_2x_2)c, \\ x_3a - y_3d + b &= (y_3x_3)c. \end{aligned} \tag{12}$$

Since p_1, p_2, and p_3 are not on a line we know $c \neq 0$. We also know that

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \neq 0. \tag{13}$$

Since $c \neq 0$,

$$f_A(x) = \frac{(\frac{a}{c})x + (\frac{b}{c})}{x + (\frac{d}{c})} = \frac{a'x + b'}{x + d'}.$$

It suffices to show that a', b', and d' are unique. Dividing (12) by c, we get

$$\begin{aligned} x_1a' - y_1d' + b' &= y_1x_1, \\ x_2a' - y_2d' + b' &= y_2x_2, \\ x_3a' - y_3d' + b' &= y_3x_3. \end{aligned}$$

The fact that this has a unique solution (which we can find) follows from equation (13).

We now prove Property 2.1 (5). Let $B = \left(\begin{smallmatrix} e & f \\ g & h \end{smallmatrix}\right)$. First consider those x's for which $gx + h = 0$ (i.e., $f_B(x) = \infty$ or ERROR). In this case we have

$$\begin{aligned} f_{AB}(x) &= f_{\left(\begin{smallmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{smallmatrix}\right)}(x) \\ &= f_{\left(\begin{smallmatrix} ae & af \\ ce & cf \end{smallmatrix}\right)}(x). \end{aligned}$$

If $ex + f = 0$ then $f_{AB}(x) =$ ERROR and $f_B(x) =$ ERROR, implying $(f_A \circ f_B)(x) =$ ERROR. If $ex + f \neq 0$, then it is easy to verify that both f_{AB} and $(f_A \circ f_B)(x)$ equal

$$\begin{cases} \frac{a}{c} & c \neq 0, \\ \infty & a \neq 0, c = 0, \\ \text{ERROR} & a = c = 0. \end{cases}$$

In the case where $gx + h \neq 0$,

$$\begin{aligned} (f_A \circ f_B)(x) &= \frac{a(\frac{ex+f}{gx+h}) + b}{c(\frac{ex+f}{gx+h}) + d} \\ &= \frac{(ae + bg)x + (af + hb)}{(ce + gd)x + (cf + hd)} \\ &= f_{\left(\begin{smallmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{smallmatrix}\right)}(x) \\ &= f_{AB}(x). \end{aligned}$$

From this we prove (Property 2.1 (6)) that $f_{A^{-1}} \equiv f_A^{-1}$ by observing that

$$(f_A \circ f_{A^{-1}})(x) = f_{AA^{-1}}(x) = f_I(x) = x$$

(where I is the identity matrix).

To prove Property 2.1 (4) we show that $f_D(x) = x$ implies that $D = \lambda I$. This suffices because the equivalence of f_A and f_B means that $f_{A^{-1}B}(x) = x$ (Property 2.1 (6)), and showing $D = A^{-1}B = \lambda I$ implies that $B = \lambda A$ as claimed. Suppose $f_D(x) = x$. If

$$D = \begin{pmatrix} d_1 & d_2 \\ d_3 & d_4 \end{pmatrix}$$

then

$$\frac{d_1x + d_2}{d_3x + d_4} = x.$$

This implies that for all x,

$$d_1x + d_2 = d_3x^2 + d_4x.$$

This can only be true if $d_2 = d_3 = 0$ and $d_1 = d_4$. Therefore

$$D = \begin{pmatrix} d_1 & 0 \\ 0 & d_4 \end{pmatrix} = d_1 I.$$

Finally, we prove Property 2.1 (7) (using Property 2.1 (5)):

$$f_A(\lambda x) = f_A\left(f_{\left(\begin{smallmatrix}\lambda & 0\\ 0 & 1\end{smallmatrix}\right)}(x)\right) = f_{A\left(\begin{smallmatrix}\lambda & 0\\ 0 & 1\end{smallmatrix}\right)}(x),$$

$$f_A(\lambda + x) = f_A\left(f_{\left(\begin{smallmatrix}1 & \lambda\\ 0 & 1\end{smallmatrix}\right)}(x)\right) = f_{A\left(\begin{smallmatrix}1 & \lambda\\ 0 & 1\end{smallmatrix}\right)}(x),$$

$$\lambda f_A(x) = f_{\left(\begin{smallmatrix}\lambda & 0\\ 0 & 1\end{smallmatrix}\right)}(f_A(x)) = f_{\left(\begin{smallmatrix}\lambda & 0\\ 0 & 1\end{smallmatrix}\right)A}(x),$$

$$\lambda + f_A(x) = f_{\left(\begin{smallmatrix}1 & \lambda\\ 0 & 1\end{smallmatrix}\right)}(f_A(x)) = f_{\left(\begin{smallmatrix}1 & \lambda\\ 0 & 1\end{smallmatrix}\right)A}(x). \qquad \square$$

Proof of Property 2.2. We first show that when the formula f is expanded to form p/q, it will be true that $\gcd(p, q) = 1$ (i.e., no simplification by nonconstant factors is ever possible). This is shown by induction on the number of nodes in f. If the gate at the root of f computes a unary function $f = f_A(g)$, where $g = p_1/q_1$, then $f = (ap_1 + bq_1)/(cp_1 + dq_1)$ (where $ad - bc \neq 0$) and $\gcd(ap_1 + bq_1, cp_1 + dq_1) = \gcd(p_1, q_1)$, which is 1 by the inductive hypothesis. If the gate at the root is + then $f = p_1/q_1 + p_2/q_2 = (p_1q_2 + p_2q_1)/(q_1q_2)$. Let $\lambda = \gcd((p_1q_2 + p_2q_1), (q_1q_2))$. Then $\lambda | q_1q_2$ and $\lambda | p_1q_2 + p_2q_1$ and, therefore,

$$\lambda | q_1(p_1q_2 + p_2q_1) - p_1q_1q_2 = p_2q_1^2$$

and

$$\lambda | q_2(p_1q_2 + p_2q_1) - p_2q_1q_2 = p_1q_2^2.$$

Since the variables in p_1 and q_1 are distinct from the variables in p_2 and q_2, we must have $\lambda = \lambda_1\lambda_2$, where $\lambda_1 | p_1$, $\lambda_1 | q_1^2$, $\lambda_2 | p_2$, and $\lambda_2 | q_2^2$. Since, by the inductive hypothesis, $\gcd(p_1, q_1) = \gcd(p_2, q_2) = 1$, we have $\lambda_1 = \lambda_2 = 1$ and $\lambda = 1$.

If the gate at the root is $\times$ then the result is obvious.

Parts (1)–(4) in this property can be proved by induction on the number of nodes in f. These claims are easily seen to be true for a single node formula; let us suppose they are true for all subformulas of f. We show only the proof of part (4). The other parts follow in a similar manner. If the gate at the root of f is a unary gate then $f = f_A(g)$, where $g = p_1/q_1$ and $f = (ap_1 + bq_1)/(cp_1 + dq_1)$. If $f(v) = \text{ERROR}$ then $g(v) = \text{ERROR}$, and, by the inductive hypothesis, $p_1(v) = q_1(v) = 0$. Then $p(v) = q(v) = 0$. If, conversely, $p(v) = q(v) = 0$ then, since A is nonsingular, $p_1(v) = q_1(v) = 0$. By the inductive hypothesis $g(v) = \text{ERROR}$ and, therefore, $f(v) = f_A(\text{ERROR}) = \text{ERROR}$.

If the gate at the root is + then $f = g_1 + g_2 = p_1/q_1 + p_2/q_2 = p/q$, where $p = p_1q_2 + p_2q_1$ and $q = q_1q_2$. If $f(v) = \text{ERROR}$ then either $g_1(v) = \text{ERROR}$, $g_2(v) = \text{ERROR}$, or $g_1(v) = g_2(v) = \infty$. If $g_1(v) = \text{ERROR}$ then by the inductive hypothesis $p_1(v) = q_1(v) = 0$ and, therefore, $p(v) = q(v) = 0$. If $g_2(v) = \text{ERROR}$ then in the same manner we get $p(v) = q(v) = 0$. If $g_1(v) = g_2(v) = \infty$ then $q_1(v) = q_2(v) = 0$ and therefore $p(v) = q(v) = 0$. On the other hand, if $p(v) = q(v) = 0$ then we have one of the following cases:

(1) $q_1(v) = q_2(v) = 0$.
(2) $q_1(v) = p_1(v) = 0$ and $q_2(v) \neq 0$.
(3) $q_2(v) = p_2(v) = 0$ and $q_1(v) \neq 0$.

This implies that either $g_1(v) =$ ERROR, $g_2(v) =$ ERROR, or $g_1(v) = g_2(v) = \infty$, and in all cases $f(v) =$ ERROR.

The case where the gate of the root is $\times$ is handled similarly. □

Proof of Property 2.3. We substitute $x_1 = f_{B^{-1}}(y_1)$ and $x_2 = f_{C^{-1}}(y_2)$ and get

$$f_{D^{-1}A}(y_1 y_2) = f_{EB^{-1}}(y_1) + f_{FC^{-1}}(y_2).$$

By substituting $y_1 = \infty$ we get $f_{D^{-1}A}(\infty) = f_{EB^{-1}}(\infty) + f_{FC^{-1}}(y_2)$ (for $y_2 \neq 0$). Since the formula on the left side is independent of y_2 it must be true that $f_{EB^{-1}}(\infty) = \infty$ (otherwise the formula on the right side will depend on y_2). By substituting $y_1 = 0$ we get $f_{D^{-1}A}(0) = f_{EB^{-1}}(0) + f_{FC^{-1}}(y_2)$ (for $y_2 \neq \infty$). Using the same arguments as before we get $f_{EB^{-1}}(0) = \infty$. Since $f_{EB^{-1}}(x)$ is bijective and $f_{EB^{-1}}(\infty) = f_{EB^{-1}}(0) = \infty$, we have a contradiction. □

Proof of Property 2.4. We first prove part (1). The "if" direction is easily verified by substitution. To prove the "only if" direction, we substitute $x_1 = f_{A_1^{-1}}(y_1)$ and $x_2 = f_{B_1^{-1}}(y_2)$ and get

$$f_{C_2^{-1}C_1}(y_1 y_2) = f_{A_2 A_1^{-1}}(y_1) f_{B_2 B_1^{-1}}(y_2).$$

As in the proof of Property 2.3, by substituting $y_1 = 0$ we get that either $f_{C_2^{-1}C_1}(0) = f_{A_2A_1^{-1}}(0) = 0$ or $f_{C_2^{-1}C_1}(0) = f_{A_2A_1^{-1}}(0) = \infty$. By using the substitutions $y_2 = 0$, $y_1 = \infty$, and $y_2 = \infty$ we conclude that we either have

$$f_{C_2^{-1}C_1}(0) = f_{A_2A_1^{-1}}(0) = f_{B_2B_1^{-1}}(0) = 0 \text{ and}$$
$$f_{C_2^{-1}C_1}(\infty) = f_{A_2A_1^{-1}}(\infty) = f_{B_2B_1^{-1}}(\infty) = \infty$$

or

$$f_{C_2^{-1}C_1}(0) = f_{A_2A_1^{-1}}(0) = f_{B_2B_1^{-1}}(0) = \infty \text{ and}$$
$$f_{C_2^{-1}C_1}(\infty) = f_{A_2A_1^{-1}}(\infty) = f_{B_2B_1^{-1}}(\infty) = 0.$$

In the first case we have for some constants $\frac{1}{\gamma}$, α, β that $f_{C_2^{-1}C_1}(x) = \frac{1}{\gamma}x$, $f_{A_2A_1^{-1}}(x) = \alpha x$, and $f_{B_2B_1^{-1}}(x) = \beta x$, and in the second case we have $f_{C_2^{-1}C_1}(x) = \frac{1}{\gamma x}$, $f_{A_2A_1^{-1}}(x) = \frac{\alpha}{x}$, and $f_{B_2B_1^{-1}}(x) = \frac{\beta}{x}$. It immediately follows that $\alpha\beta\gamma = 1$, which gives us the the desired result.

The proof of part (2) is similar. □

Proof of Property 2.5. To prove part (1), we substitute $x_1 = f_{(A_1')^{-1}}(0)$ and get

$$f_C(f_{A_3}(x_3) \times f_B(f_{A_1(A_1')^{-1}}(0) \times f_{A_2}(x_2))) = f_{C'}(0).$$

Since the formula is independent of x_2 and x_3, $f_{A_1(A_1')^{-1}}(0) \in \{0, \infty\}$ and either $f_B(0)$ or $f_B(\infty)$ is equal to 0 or ∞.

By substituting $x_1 = f_{(A_1')^{-1}}(\infty)$ and using the fact that f_B is bijective we conclude that

$$\{f_B(0), f_B(\infty)\} = \{0, \infty\}.$$

This implies that

$$B = \begin{pmatrix} \alpha_1 & 0 \\ 0 & \alpha_2 \end{pmatrix} \text{ or } B = \begin{pmatrix} 0 & \alpha_1 \\ \alpha_2 & 0 \end{pmatrix}.$$

The proof of the second part is similar. □

Acknowledgments. We thank Joachim von zur Gathen for bringing to our attention the work of Heintz and Schnorr [13].

REFERENCES

[1] D. ANGLUIN, *Queries and concept learning*, Machine Learning, 2 (1988), pp. 319–342.

[2] D. ANGLUIN, L. HELLERSTEIN, AND M. KARPINSKI, *Learning read-once formulas with queries*, J. Assoc. Comput. Mach., 40 (1993), pp. 185–210.

[3] M. BEN-OR AND P. TIWARI, *A deterministic algorithm for sparse multivariate polynomial interpolation*, in Proc. 20th Annual ACM Symposium on the Theory of Computing, Chicago, 1988, pp. 301–309.

[4] A. BORODIN AND P. TIWARI, *On the decidability of sparse univariate polynomials*, in Proc. 31st Symposium on the Foundations of Computer Science, IEEE, St. Louis, MO, 1990.

[5] N. H. BSHOUTY, T. HANCOCK, AND L. HELLERSTEIN, *Learning boolean read-once formulas with arbitrary symmetric and constant fan-in gates*, in Proc. 5th Annual ACM Workshop on Computational Learning Theory, Pittsburgh, PA, ACM, 1992; J. Comput. System Sci., to appear.

[6] N. H. BSHOUTY, T. R. HANCOCK, L. HELLERSTEIN, AND M. KARPINSKI, *An algorithm to learn read-once threshold formulas, and transformations between learning models*, Comput. Complexity, to appear.

[7] S. GOLDMAN, M. KEARNS, AND R. SCHAPIRE, *Exact identification of read-once formulas using fixed points of amplification functions*, SIAM J. Comput., 22 (1993), pp. 705–726.

[8] D. Y. GRIGORIEV, M. KARPINSKI, AND M. SINGER, *Computational complexity of sparse rational interpolation*, SIAM J. Comput., 23 (1994), pp. 1–11.

[9] ———, *Fast parallel algorithms for sparse multivariate polynomial interpolation over finite fields*, SIAM J. Comput., 19 (1990), pp. 1059–1063.

[10] T. HANCOCK, *Identifying μ-formula decision trees with queries*, Tech. report TR-16-90, Harvard University, Computer Science Dept., Cambridge, MA, 1990.

[11] T. HANCOCK AND L. HELLERSTEIN, *Learning read-once formulas over fields and extended bases*, in Proc. 4th Annual Workshop on Computational Learning Theory, Santa Cruz, CA, ACM, SIGACT/SIGART, 1991, pp. 326–336.

[12] T. R. HANCOCK, M. GOLEA, AND M. MARCHAND, *Learning nonoverlapping perceptron networks from examples and membership queries*, Tech. report TR-26-91, Harvard University, Computer Science Dept., Cambridge, MA, 1991; Machine Learning, to appear.

[13] J. HEINTZ AND C. P. SCHNORR, *Testing polynomials that are easy to compute*, in Proc. 12th Annual ACM Symposium on the Theory of Computing, Los Angeles, 1980, pp. 262–272.

[14] B. LHOTZKY, *On the computational complexity of some algebraic counting problems*, Ph.D. thesis, University of Bonn, Computer Science Dept., Bonn, Germany, 1991.

[15] M. RON ROTH AND G. BENEDEK, *Interpolation and approximation of sparse multivariate polynomials over $gf(2)$*, SIAM J. Comput., 20 (1991), pp. 291–314.

[16] J. T. SCHWARTZ, *Fast polynomial algorithms for verification of polynomial identities*, J. Assoc. Comput. Mach., 27 (1980), pp. 701–707.

SIAM J. COMPUT.
Vol. 24, No. 4, pp. 736–750, August 1995

004

AMORTIZED COMMUNICATION COMPLEXITY*

TOMÀS FEDER†, EYAL KUSHILEVITZ‡, MONI NAOR§, AND NOAM NISAN¶

Abstract. In this work we study the *direct-sum* problem with respect to communication complexity: Consider a relation f defined over $\{0,1\}^n \times \{0,1\}^n$. Can the communication complexity of simultaneously computing f on ℓ instances $(x_1, y_1), \ldots, (x_\ell, y_\ell)$ be smaller than the communication complexity of separately computing f on the ℓ instances?

Let the *amortized* communication complexity of f be the communication complexity of simultaneously computing f on ℓ instances divided by ℓ. We study the properties of the amortized communication complexity. We show that the amortized communication complexity of a relation can be smaller than its communication complexity. More precisely, we present a *partial function* whose (deterministic) communication complexity is $\Theta(\log n)$ and amortized (deterministic) communication complexity is $O(1)$. Similarly, for *randomized* protocols we present a function whose randomized communication complexity is $\Theta(\log n)$ and amortized randomized communication complexity is $O(1)$.

We also give a general lower bound on the amortized communication complexity of any *function* f in terms of its communication complexity $C(f)$: for every function f the amortized communication complexity of f is $\Omega(\sqrt{C(f)} - \log n)$.

Key words. communication complexity, graph coloring, simultaneous computation

AMS subject classifications. 94A05, 68Q22, 68R99

1. Introduction.

A very basic question in the theory of computation is the *direct-sum* question: Can the cost of solving ℓ independent instances of a problem simultaneously be smaller than the cost of independently solving the ℓ problems, say, sequentially? In this work we study the direct-sum question in the context of communication complexity. This question was recently raised by Karchmer, Raz, and Wigderson [7] as part of a new approach for proving lower bounds on Boolean circuits using communication complexity arguments (as in [8] and [19]). For a general survey on communication complexity see [12]. Different scenarios where the direct-sum question was investigated are [4], [6], [18], and [21].

Let f be a *relation* defined on $\{0,1\}^n \times \{0,1\}^n$.[1] Let $f^{(\ell)}$ be the extension of f to ℓ instances. The communication complexity problem associated with $f^{(\ell)}$ is as follows: Party P_1 receives ℓ inputs $x_1, \ldots, x_\ell$ and party P_2 receives ℓ inputs $y_1, \ldots, y_\ell$ (each of x_i and y_i is an n bit string). They need to find values $z_1, \ldots, z_\ell$ such that for each i, the value z_i satisfies the relation $f(x_i, y_i)$. Denote by $C(f)$ the communication complexity of f, namely, the number of bits that the parties need to exchange on the worst-case input in the best protocol for computing f. Similarly, denote by $\overline{C}(f)$ the amortized communication complexity of f,

*Received by the editors August 10, 1992; accepted for publication (in revised form) January 27, 1994. An early version of this paper appeared in Proc. 32nd IEEE Conference on the Foundations of Computer Science, October 1991, pp. 239–248.

†IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120.

‡Computer Science Department, Technion–Israel Institute of Technology, Haifa 32000, Israel (eyalk@cs.technion.ac.il). This research was supported in part by U.S.–Israel Binational Scientific Fund grant 88-00282. Some of this work was done while the author was at the Aiken Computation Laboratory, Harvard University.

§Department of Applied Mathematics and Computer Science, Weizmann Institute of Science, Rehovot 76100, Israel (naor@wisdom.weizmann.ac.il). Most of this work was done while the author was at the IBM Almaden Research Center.

¶Computer Science Department, Hebrew University of Jerusalem, Jerusalem 91904, Israel (noam@cs.huji.ac.il). This research was supported by the Wolfson research awards administered by the Israel Academy of Sciences and Humanities and by U.S.–Israel Binational Scientific Fund grant 89-00126.

[1] A *relation* defines a subset $f(x, y)$ of a domain $\mathcal{D}$ for every input pair $(x, y) \in \{0,1\}^n \times \{0,1\}^n$. In particular, we will be interested in two special cases of relations: *functions*, where for each input pair (x, y) there is a unique value in $f(x, y)$, and *partial functions*, where for each input pair (x, y) there is either a unique possible value or all values in $\mathcal{D}$ are possible.

namely,

$$\overline{C}(f) = \limsup_{\ell \to \infty} \frac{1}{\ell} C(f^{(\ell)}).$$

Clearly, $\overline{C}(f) \leq C(f)$ for every relation f. It was observed in [7] that when (nonpartial) functions are considered, an upper bound on $\overline{C}(f)$, which is significantly smaller than $C(f)$, implies that the *rank* lower bound on $C(f)$ [13] is not tight. This is true because the rank of the matrix representing $f^{(\ell)}$ equals the rank of the matrix representing f to the power of ℓ.

We present a *partial* function f such that $C(f) = \Theta(\log n)$ and $\overline{C}(f) = O(1)$. This proves that computing a relation f on ℓ instances simultaneously may be easier than computing f on the ℓ instances separately. In [7] it was conjectured that $\overline{C}(f)$ cannot be smaller than $C(f)$ by more than an *additive* factor of $O(\log n)$. We prove two weaker versions of this conjecture:

- If *one-way* communication protocols are considered then *any* partial function f over $\{0,1\}^n \times \{0,1\}^n$ satisfies $\overline{C}_1(f) \geq C_1(f) - \log n - O(1)$.
- For *general* (two-way) protocols, any (nonpartial) function f over $\{0,1\}^n \times \{0,1\}^n$ satisfies $\overline{C}(f) \geq \sqrt{C(f)/2} - \log n - O(1)$.

The proof of the first lower bound is via a reduction to an appropriate graph-coloring problem followed by application of the results of Linial and Vazirani [11] on the chromatic number of product graphs. The lower bound for general protocols is achieved by considering *nondeterministic* protocols and proving that $\overline{C}_N(f) \geq C_N(f) - \log n - O(1)$, then applying a result of Aho, Ullman, and Yannakakis [1] which relates the nondeterministic communication complexity of a function with its deterministic communication complexity.

We also study the direct-sum question with respect to *randomized* protocols. The only trivial upper bound on $C_R(f^{(\ell)})$ in this case is that for any (partial or nonpartial) function f, $C_R(f^{(\ell)}) = O(\ell \cdot \log \ell \cdot C_R(f))$ (the $\log \ell$ factor seems to be needed, since we are required to have a "good" probability of success on *all* ℓ instances simultaneously). For explicit functions we can do much better: We consider the *identity* function (i.e., $ID : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ defined by $ID(x, y) = 1$ iff $x = y$). It is well known that $C_R(ID) = \Theta(\log n)$ [24]. We prove that $\overline{C}_R(ID) = O(1)$.

Remark. Some of the results of this paper were re-proven in a completely different way in [10].

The organization of the paper is as follows: In §2 the various notions of communication complexity and amortized communication complexity are defined. In §3 we exhibit a partial function whose amortized communication complexity is smaller than its communication complexity. In §4 we discuss the special case of one-way communication protocols. In §5 we prove our lower bound on the amortized communication complexity for the case of general protocols. In §6 we present a function whose amortized communication complexity is smaller than its communication complexity when randomized protocols are considered. Finally, in §7 we mention some open problems.

2. Preliminaries. In this section we give formal definitions for the various notions of communication protocols and communication complexity used in this work.

Let $\mathcal{D}$ be a set, and let f be a *relation* defined over $\{0,1\}^n \times \{0,1\}^n$ such that for every $(x, y) \in \{0,1\}^n \times \{0,1\}^n$, it satisfies $\emptyset \neq f(x, y) \subseteq \mathcal{D}$. We say that f is *Boolean* if $\mathcal{D} = \{0,1\}$. We say that f is a *function* if for every (x, y), $|f(x, y)| = 1$, and f is a *partial function* if for every (x, y), either $|f(x, y)| = 1$ or $f(x, y) = \mathcal{D}$.

Given a relation f and an integer $\ell \geq 1$, we define the relation $f^{(\ell)}$ over $(\{0,1\}^n)^\ell \times (\{0,1\}^n)^\ell$ with range $\mathcal{D}^\ell$ as follows:

$$f^{(\ell)}((x_1,\ldots,x_\ell),(y_1,\ldots,y_\ell)) \stackrel{\Delta}{=} \{(z_1,\ldots,z_\ell) \mid z_1 \in f(x_1,y_1),\ldots,z_\ell \in f(x_\ell,y_\ell)\}.$$

In what follows we define the communication complexity of relations of the form $f^{(\ell)}$. Note, however, that this covers the special case of $f^{(1)} \equiv f$.

Two parties P_1 and P_2 wish to compute a possible value of $f^{(\ell)}$ on their input. The party P_1 is given an $n\ell$-bit input x and the party P_2 is given an $n\ell$-bit input y. We interpret x (resp., y) as consisting of ℓ pieces (or *instances*) $x_1,\ldots,x_\ell$ (resp., $y_1,\ldots,y_\ell$) each of n bits. The parties exchange messages in rounds according to a *deterministic* protocol. That is, each message sent by a party P_i depends on its input and the messages it received in previous rounds. The last message in the protocol is an ℓ-tuple $z = (z_1,\ldots,z_\ell)$ called the *output* of the protocol. We say that a protocol $\mathcal{F}$ computes the relation $f^{(\ell)}$ if, for all inputs x and y, the output z satisfies $z \in f^{(\ell)}(x,y)$.

The concatenation of all the messages exchanged in the protocol $\mathcal{F}$ on input (x,y) is denoted $\mathcal{F}(x,y)$. The (*deterministic*) communication complexity of the protocol $\mathcal{F}$, denoted $C(\mathcal{F})$, is the maximum $|\mathcal{F}(x,y)|$ over all (x,y). The (*deterministic*) communication complexity of the relation $f^{(\ell)}$, denoted $C(f^{(\ell)})$, is the minimum of $C(\mathcal{F})$ over all deterministic protocols $\mathcal{F}$ computing $f^{(\ell)}$.

The *amortized communication complexity* of the relation f is defined as

$$\overline{C}(f) = \limsup_{\ell\to\infty} \frac{1}{\ell} C(f^{(\ell)}).$$

We sometimes restrict the discussion to *one-way* protocols. In such protocols the communication consists of a single message: P_1 sends a message to P_2 and P_2 has to compute the output. We denote by $C_1(\mathcal{F})$, $C_1(f)$, and $\overline{C}_1(f)$ the analogues of $C(\mathcal{F})$, $C(f)$, and $\overline{C}(f)$ for the case in which only one-way protocols are considered.

We also consider *randomized* protocols, in which each of the parties has, in addition to its input, a string of random coins (the random strings of the two parties are independent). A randomized protocol $\mathcal{F}$ computes the relation $f^{(\ell)}$ if, for every input (x,y) the output z of $\mathcal{F}$ satisfies $z \in f^{(\ell)}(x,y)$ with probability $\geq \frac{3}{4}$. The notions of $C_R(\mathcal{F})$, $C_R(f^{(\ell)})$, and $\overline{C}_R(f)$ are defined in a similar way with respect to randomized protocols. That is, $C_R(\mathcal{F})$ is the maximal length of communication (over all inputs and all strings of random coins) in the protocol $\mathcal{F}$, $C_R(f^{(\ell)})$ is the minimum of $C_R(\mathcal{F})$ over all randomized protocols that compute the relation $f^{(\ell)}$, and $\overline{C}_R(f)$ equals $\limsup_{\ell\to\infty} \frac{1}{\ell} C_R(f^{(\ell)})$. We emphasize that the meaning of this definition is that when computing $f^{(\ell)}$ we require that with probability at least $\frac{3}{4}$ the output is correct for *all* ℓ instances simultaneously.

It is also useful to consider a variant of the randomized model in which both parties have access to a *public* random string. The quantities $C_{\text{pub}}(f^{(\ell)})$ and $\overline{C_{\text{pub}}}(f)$ are defined in a similar way.

Finally, we give the definitions for the *nondeterministic* case. In a nondeterministic protocol for computing $f^{(\ell)}$, the parties are allowed to make "guesses" while choosing their messages. In any computation, the protocol gives either a correct value of $f^{(\ell)}(x,y)$ or "fail". The protocol is required to output a correct value of $f^{(\ell)}(x,y)$ in at least one computation on (x,y) (i.e., in this computation the output is correct for *all* ℓ instances). The nondeterministic complexity of a protocol $\mathcal{F}$, $C_N(\mathcal{F})$, is defined as the maximum over all (x,y) and over all computations ("guesses") of $\mathcal{F}(x,y)$ (note that for nondeterministic protocols $\mathcal{F}(x,y)$ is not unique). The measures $C_N(f^{(\ell)})$ and $\overline{C}_N(f)$ are defined with respect to nondeterministic protocols.

3. A partial function with a low amortized complexity. In this section we prove that (deterministic) amortized communication complexity can be substantially lower than the corresponding communication complexity. We present a partial function f such that $C(f) = \Theta(\log n)$, while $\overline{C}(f) = O(1)$.

We start with the definition of f: Let $M = \{0, 1, 2, \ldots, m-1\}$. Let $t \geq 2$ be a parameter. The input of P_1 is S, a subset of M of size t (the length of this input is $n = t \cdot \log m$ bits). The input of P_2 is $x \in S$ (the length of this input is $\log m < n$ bits). The parties wish to compute the rank of x in the subset S (a number in the range $0, \ldots, t-1$). If $x \notin S$ then any output (in the range $0, \ldots, t-1$) is allowed. Orlitsky [17] showed that the communication complexity of this function is $C(f) = \Theta(\log t + \log\log m)$.

The protocols we present make use of the following set of hash functions suggested by Fredman, Komlòs, and Szemerèdi [5]: Let $p \simeq t^2 \log m$ be a prime. Define

$$H = \left\{h : M \to \{0, 1, \ldots, 2t^2 - 1\} \| \; h(x) = (ax \bmod p) \bmod 2t^2, \;\; 1 \leq a \leq p-1\right\}.$$

We say that $h \in H$ is *good* for a set $S \subset M$ if h is $1-1$ with respect to the elements of S. Otherwise, we say that h is *bad* for S. Fredman, Komlòs, and Szemerèdi [5] proved the following property of these hash functions.

LEMMA 3.1. *Let H be as above and let S be any subset of M of size t. Then, at least $\frac{1}{2}$ of the functions in H are good for S.*

We start by presenting the following protocol from [17] that meets the lower bound for computing f on a *single* instance (S, x). This protocol (which uses the above H) has the advantage that an appropriate generalization of it gives the amortized result.

- P_1 finds a function $h \in H$, which is good with respect to S. It sends its name ($O(\log t + \log\log m)$ bits) to P_2.
- P_2 computes $h(x)$ and sends this value ($O(\log t)$ bits) to P_1.
- Since h is good with respect to S, if $x \in S$, the value $h(x)$ determines x. (If $x \notin S$ then either $h(x) = h(s)$ for some $s \in S$ or it does not. For the correctness of the protocol it does not matter which is the case.) Now P_1 computes the value $f(S, x)$ and sends it to P_2 ($O(\log t)$ bits).

We now show how to generalize the protocol in order to efficiently compute the values $f(S_1, x_1)$, $f(S_2, x_2), \ldots, f(S_\ell, x_\ell)$ simultaneously. The main idea is formalized by the following claim.

CLAIM 3.2. *Let H be as above and let $S_1, \ldots, S_\ell$ be any ℓ subsets of M of size t. Then there exists a set L of $\log \ell + 1$ hash functions $h_1, h_2, \ldots, h_{\log \ell + 1} \in H$ such that*

- *for every j $(1 \leq j \leq \log \ell + 1)$, h_j is good with respect to at least $\frac{1}{2}$ of the S_i's for which $h_1, \ldots, h_{j-1}$ are all bad.*

In particular, it follows that for every S_i $(1 \leq i \leq \ell)$, there exists at least one hash function in L, denoted $h_{j(i)}$, such that $h_{j(i)}$ is good for S_i. The proof uses Lemma 3.1 and a simple counting argument.

Proof. We show how to construct L iteratively. In the jth iteration we consider a matrix with all the subsets S_i for which $h_1, \ldots, h_{j-1}$ are bad as rows, and the hash functions in H as columns. The (S, h) entry in this matrix is 1 if h is good with respect to S, and 0 otherwise. By Lemma 3.1, at least half of the entries in every row are 1's. Therefore, there exists a column in which at least half of the entries are 1's. We take the corresponding hash function as h_j. □

The following protocol computes f on ℓ instances simultaneously:

- P_1 finds a set L of $\log \ell + 1$ hash functions as above and sends the names of functions in L to P_2. In addition, for every $1 \leq i \leq \ell$, it sends the index $j(i)$.
- P_2 computes $h_{j(i)}(x_i)$ for every i and sends it to P_1.

- Since $h_{j(i)}$ is good with respect to S_i, the party P_1 knows the value of x_i for every $1 \leq i \leq \ell$ and thus can compute $f(S_1, x_1), \ldots, f(S_\ell, x_\ell)$.

The correctness of the protocol is obvious. For every i such that $x_i \in S_i$ it computes the correct answer (and if $x_i \notin S_i$ then any answer is good). We now analyze its complexity.

CLAIM 3.3. *The above protocol can be implemented so that the number of bits exchanged is* $O(\ell \cdot \log t + \log \ell \cdot (\log t + \log \log m))$.

Proof. To specify the names of functions in L, P_1 uses $O(\log \ell \cdot (\log t + \log \log m))$ bits. In addition, for specifying *all* the indices $j(i)$, P_1 needs only $O(\ell)$ bits (which is better than the obvious $O(\ell \log \ell)$ bits). This is true because h_1 is good for about $\frac{1}{2}$ of the sets, h_2 is good for about $\frac{1}{4}$ of the sets, etc. Therefore, by using, say, Huffman coding, we get that $O(\ell)$ bits are enough. In the second step P_2 sends the results of applying $h_{j(i)}$ on x_i for every i, which requires $O(\ell \cdot \log t)$ bits. □

Take, for example, $t = 2$ and recall that in this case the length of the input satisfies $n = 2 \log m$. We get that the number of bits exchanged in this protocol is $O(\ell + \log \ell \cdot \log n)$. Thus, we proved the following theorem.

THEOREM 3.4. *There exists a (partial) function* $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ *with communication complexity* $C(f) = \Theta(\log n)$ *and amortized communication complexity*

$$\overline{C}(f) = \limsup_{\ell \to \infty} \frac{1}{\ell} C(f^{(\ell)}) = O(1).$$

4. One-way communication. In this section we deal with one-way communication protocols. We show that if we restrict the discussion to the computation of relations using one-way protocols, then we can still "save" bits by computing f on many instances simultaneously. In fact, the partial function f of the previous section yields such an example: Take $t = 2$ and assume that $S_i = \{y_1^i, y_2^i\}$, where $0 \leq y_1^i < y_2^i \leq m - 1$. As stated before, $C(f) = \Theta(\log n)$ (and clearly $C_1(f) \geq C(f)$). On the other hand, a slight modification of the previous protocol gives $\overline{C}_1(f) = O(1)$: P_1 sends $h_{j(i)}(y_1^i)$ and $h_{j(i)}(y_2^i)$ for $1 \leq i \leq \ell$ together with the list L of hash functions. Now P_2 can decide whether $x_i = y_1^i$ or $x_i = y_2^i$.

On the other hand, we can prove that for every (partial) function f, no more than $\log n$ bits can be saved: $\overline{C}_1(f) \geq C_1(f) - \log n - O(1)$. We start with a simple theorem, which claims that if f is a *nonpartial* function then essentially nothing can be saved. That is, $\overline{C}_1(f) \cong C_1(f)$.

THEOREM 4.1. *Let* f *be a (nonpartial) function defined on* $\{0,1\}^n \times \{0,1\}^n$. *Then* $C_1(f) - 1 \leq \overline{C}_1(f) \leq C_1(f)$.

Proof. Define the following relation on the inputs of P_1: $x_1 \sim x_2$ if $f(x_1, y) = f(x_2, y)$ for every y. Clearly $\sim$ is an equivalence relation. Denote by $Class(f)$ the number of equivalence classes of the $\sim$ relation. It can be easily verified that for computing f the party P_1 must use $Class(f)$ different messages (i.e, $C_1(f)$ is exactly $\lceil \log Class(f) \rceil$). This is true since P_1 can send, on input x, the index of equivalence class for which x belongs. From this information P_2 can easily compute $f(x, y)$ (by choosing arbitrary x' from that equivalence class and computing $f(x', y)$). On the other hand, if for two inputs x, x' in different equivalence classes P_1 sends the same string, then by the definition of the relation $\sim$ there exists y such that $f(x, y) \neq f(x', y)$. If P_2 holds y as his input then clearly the protocol is wrong for at least one of $f(x, y)$ or $f(x', y)$. Similar arguments show that for computing $f^{(\ell)}$ the party P_1 must use $Class(f^{(\ell)}) = Class(f)^\ell$ different messages. Since this number of strings is enough, the theorem follows. □

The above example shows that this result cannot be extended to *partial* functions. The key point is that for partial functions $\sim$ is not necessarily an equivalence relation. However,

in the following we show that this example is optimal in a sense. More precisely, we prove for every partial function f that $\overline{C}_1(f)$ cannot be smaller than $C_1(f)$ by more than an additive factor of $O(\log n)$.

THEOREM 4.2. *Let f be a (partial) function defined over $\{0,1\}^n \times \{0,1\}^n$. Then $C_1(f^{(2)}) \geq 2C_1(f) - \log n - O(1)$.*

Proof. The idea of the proof is to reduce the problem of the one-way communication complexity of a function to an appropriate graph-coloring problem,[2] and then to use results of Linial and Vazirani [11] on this problem.

We construct a graph $G_f = (V, E)$ as follows: Each vertex corresponds to $x \in \{0,1\}^n$. There is an edge between x and x' if there exists y such that $f(x, y) \cap f(x', y) = \emptyset$ (this happens if and only if $|f(x, y)| = |f(x', y)| = 1$ and $f(x, y) \neq f(x', y)$). Intuitively, there is an edge between x and x' if P_2 must be able to distinguish between these two inputs to compute the output correctly when it holds input y (since there is no output which is legal for both (x, y) and (x', y)). Similarly, we define a graph $G_{f^{(2)}}$; its vertices correspond to pairs $(x_1, x_2) \in \{0,1\}^n \times \{0,1\}^n$. There is an edge between $x = (x_1, x_2)$ and $x' = (x_1', x_2')$ if there exists $y = (y_1, y_2)$ such that $f^{(2)}(x, y) \cap f^{(2)}(x', y) = \emptyset$ (this happens if and only if either $|f(x_1, y_1)| = |f(x_1', y_1)| = 1$ and $f(x_1, y_1) \neq f(x_1', y_1)$, or if $|f(x_2, y_2)| = |f(x_2', y_2)| = 1$ and $f(x_2, y_2) \neq f(x_2', y_2)$).

The number of different messages used by the optimal one-way communication protocol for f is exactly the *chromatic number* of G_f (denoted $\chi(G_f)$). If we have a legal coloring of G_f then this coloring defines a one-way communication protocol for computing f: P_1 sends the color c of its input x. This color together with P_2's input y determine $z \in f(x, y)$. To see this, fix a y and consider all the vertices colored by c. If for all these vertices the corresponding x satisfies $f(x, y) = \mathcal{D}$, then any $z \in \mathcal{D}$ will do. If for some x, $|f(x, y)| = 1$ then we take $z = f(x, y)$. For any other x' colored by c, since there is no edge between x and x', it follows from the construction that $z \in f(x', y)$. On the other hand, every protocol induces a legal coloring of G_f, where the color of every x is the message P_1 sends on it. This is true because for every x, x' on which the same message m is sent by P_1, and for every y, there is a z that P_2 outputs. The correctness of the protocol guarantees that $z \in f(x, y)$ and $z \in f(x', y)$ and, therefore, $f(x, y) \cap f(x', y) \neq \emptyset$. Hence there is no edge between x and x', so the coloring is legal. Similarly, the number of different messages used by the optimal one-way communication protocol for $f^{(2)}$ is exactly $\chi(G_{f^{(2)}})$ (again, fix (y_1, y_2) and argue the existence of z_1 and z_2 as needed for each coordinate separately).

Now we define the *product* operation on graphs: Given $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the vertices set of the product $G_1 \times G_2$ is $V_1 \times V_2$. The edge set includes all the edges $((v_1, v_2), (u_1, u_2))$ such that $(v_1, u_1) \in E_1$ *or* $(v_2, u_2) \in E_2$. (In the terminology of [11] this is called *inclusive*-product.) It is easy to verify that $G_{f^{(2)}} = G_f \times G_f$.

Applying this reduction to the graph-coloring problem we can now prove the theorem: it is enough to prove that for every f, $\chi(G_{f^{(2)}}) \geq \chi^2(G_f)/cn$ for some constant c. This is proved in [11, Thm. 1]. □

The statement of [11, Thm. 1] is more general than what we used and allows not only products of a graph G by itself but products of any two graphs. In particular, it says that for any two graphs G_1, G_2 such that $|V_1| \leq |V_2|$, the chromatic number satisfies $\chi(G_1 \times G_2) \geq \chi(G_1)\chi(G_2)/c \log |V_1|$. Thus, by the same proof as above, we get the following theorem.

THEOREM 4.3. *Let $n \leq m$. Let f be a (partial) function defined over $\{0,1\}^n \times \{0,1\}^n$ and let g be a (partial) function defined over $\{0,1\}^m \times \{0,1\}^m$. Let $f \times g$ be defined in the*

[2] Similar reductions appear in [17] and [22]. In these works the two parties have an input (x, y) in some domain $\mathcal{A}$, and P_1 has to transmit its input x to P_2. In our setting this problem corresponds to the problem of computing the specific function f which is defined as $f(x, y) = x$ if $(x, y) \in \mathcal{A}$ and $f(x, y) = \mathcal{D}$ otherwise.

obvious way over $(\{0,1\}^n \times \{0,1\}^m) \times (\{0,1\}^n \times \{0,1\}^m)$ *(each party receives two instances; one is an n-bit string and the other is an m-bit string). Then,*

$$C_1(f \times g) \geq C_1(f) + C_1(g) - \log n - O(1).$$

Therefore, we have the following corollary.

COROLLARY 4.4. *Let f be a (partial) function defined over $\{0,1\}^n \times \{0,1\}^n$. Then*

$$C_1(f) \geq \overline{C}_1(f) \geq C_1(f) - \log n - O(1).$$

Proof. The first inequality is obvious. For the second inequality we will prove (by induction) that $C_1(f^{(\ell)}) \geq \ell C_1(f) - (\ell - 1)\log n - (\ell - 1)c$ (for some constant c), which implies the corollary. This is certainly true for $\ell = 1$. For a general ℓ we can write $C_1(f^{(\ell)}) = C_1(f \times f^{(\ell-1)})$. By Corollary 4.3 this is at least $C_1(f) + C_1(f^{(\ell-1)}) - \log n - c$. Now, by the induction hypothesis $C_1(f^{(\ell-1)}) \geq (\ell - 1)C_1(f) - (\ell - 2)\log n - (\ell - 2)c$, which gives us what we need. □

For additional examples of partial functions with $\overline{C}_1(f)$ significantly smaller than $C_1(f)$, we show that for every graph G with 2^n vertices there exists a (partial) function f such that $G = G_f$. Label the vertices of G by strings in $\{0,1\}^n$ and define a function f as follows: for every x, $f(x,x) = 1$. For every edge $(x,y) \in E$ define $f(x,y) = 0$. For all the other pairs $f(x,y) = \mathcal{D}$. It can be easily verified that $G = G_f$. This implies that, from every graph G with 2^n vertices, such that $\chi(G \times G) \cong \chi^2(G)/cn$, we can construct a partial function f such that $C_1(f^{(2)}) \cong 2C_1(f) - \log n - O(1)$. Examples of such graphs are given in [11, Thm. 2].

5. Lower bound for general protocols. In order to prove lower bounds on $\overline{C}(f)$ for a specific relation f, we may use traditional techniques. For example, consider the *identity* function (i.e., $ID(x,y)$ equals 1 if $x = y$, and 0 otherwise). It is easy to verify that $\overline{C}(ID) = C(ID) = n$ (as in [24]). In this section we give a *general* lower bound on $\overline{C}(f)$ in terms of $C(f)$ for any *(nonpartial) Boolean function* f.

To this end, we first discuss the amortized *nondeterministic* communication complexity of (arbitrary) relations. We start with some definitions and notation used in the proof. Given a relation f defined over $\{0,1\}^n \times \{0,1\}^n$, and $\ell \geq 1$, we denote by $M_{f^{(\ell)}}$ the matrix representing the relation $f^{(\ell)}$. That is, each row of $M_{f^{(\ell)}}$ corresponds to an input $x = (x_1, x_2, \ldots, x_\ell)$ of P_1, and each column corresponds to an input $y = (y_1, y_2, \ldots, y_\ell)$ of P_2. The entry (x,y) of $M_{f^{(\ell)}}$ contains the set $f(x,y)$ (a subset of $\mathcal{D}^\ell$). A *monochromatic rectangle* of $M_{f^{(\ell)}}$ is a set $R = R_x \times R_y \subseteq (\{0,1\}^n)^\ell \times (\{0,1\}^n)^\ell$ such that we can associate with R an output vector $z_R \in \mathcal{D}^\ell$ in such a way that every input $(x,y) \in R$ satisfies $z_R \in f(x,y)$. We denote by $N(f^{(\ell)})$ the minimal number of monochromatic rectangles needed to cover (possibly with overlaps) all the entries of $M_{f^{(\ell)}}$. Since any nondeterministic protocol for computing $f^{(\ell)}$ induces such a cover, $\log N(f^{(\ell)}) \leq C_N(f^{(\ell)})$. The next theorem claims that $N(f^{(2)})$ cannot be much smaller than $N^2(f)$.

THEOREM 5.1. *Let f be a relation defined over $\{0,1\}^n \times \{0,1\}^n$. Then, for some constant c,*

$$N(f^{(2)}) \geq \frac{N^2(f)}{c \cdot n}.$$

For the proof of this theorem we need the following claim, provided by the proof of [11, Thm. 1].

CLAIM 5.2. *Let A be an $\ell \times d$ matrix whose entries assume k values and such that $\ell \leq d$. Let k_1 be the minimal size of a set $T \subseteq \{1, 2, \ldots k\}$ that covers all the rows of A. That is, for*

every row i there exists a column j such that the value $A_{i,j}$ belongs to T. Similarly, let k_2 be the minimal size of a set that covers all the columns. Then $k_1 \cdot k_2 \le c' \cdot \log \ell \cdot k$.

Proof. Consider an optimal cover of $M_{f^{(2)}}$ with $k = N(f^{(2)})$ monochromatic rectangles, denoted by $R_1, R_2, \ldots, R_k$. We show how to cover M_f with m monochromatic rectangles, where $m^2 \le c \cdot n \cdot N(f^{(2)})$ for some constant c. This implies that $N^2(f) \le c \cdot n \cdot N(f^{(2)})$.

Consider the following $2^{2n} \times 2^{2n}$ matrix A (this is *not* $M_{f^{(2)}}$): each row of A corresponds to an input (x_1, y_1) and each column corresponds to an input (x_2, y_2). Every entry $((x_1, y_1), (x_2, y_2))$ of A contains an element t in $\{1, 2, \ldots k\}$ such that $((x_1, x_2), (y_1, y_2))$ belongs to R_t. (If $((x_1, x_2), (y_1, y_2))$ belongs to more than one rectangle then we choose one of them arbitrarily.) Apply Claim 5.2 to the matrix A described above and assume without loss of generality that $k_1 \le k_2$; we get that $k_1^2 \le c \cdot n \cdot k$. Let T be a set of k_1 values that covers the rows. We now prove that this implies that M_f can be covered with k_1 monochromatic rectangles.

Associate with every entry (x, y) in M_f an element of T that appears in the row (x, y) of A (if there is more than one possibility, then choose one arbitrarily). Now we extend this to (possibly overlapping) rectangles in the obvious way. That is, for every $t \in T$ the rectangle R'_t includes every (x, y) with value t, and if (x, y) and (x', y') are in R'_t then (x', y) and (x, y') are in R'_t as well.

Clearly, these are k_1 rectangles and they cover M_f. What we still have to prove is that any such rectangle R'_t is monochromatic. That is, there exists a z such that for all $(x, y) \in R'_t$ it satisfies $z \in f(x, y)$. By the construction, if (x, y) and (x', y') both have the value t, then there exist x_2, y_2, x'_2, and y'_2 such that both $((x, x_2), (y, y_2))$ and $((x', x'_2), (y', y'_2))$ belong to R_t. Since R_t is monochromatic, we can associate with R_t a vector (z_1, z_2) with whom all pairs in R_t "agree." This, in particular, implies that $z_1 \in f(x, y)$ and $z_1 \in f(x', y')$. In addition, since R_t is a rectangle it also contains $((x, x_2), (y', y'_2))$, and $((x', x'_2), (y, y_2))$, which implies that $z_1 \in f(x, y')$ and $z_1 \in f(x', y)$ as well. Therefore R'_t is monochromatic.

To conclude, we can cover M_f with no more than $\sqrt{c \cdot n \cdot N(f^{(2)})}$ monochromatic rectangles, which completes the proof of the theorem. □

Again, the above theorem (using [11]) can be generalized to prove the following.

THEOREM 5.3. *Let $n \le m$. Let f be a relation defined over $\{0,1\}^n \times \{0,1\}^n$ and let g be a relation defined over $\{0,1\}^m \times \{0,1\}^m$. Then,*

$$N(f \times g) \ge \frac{N(f) \cdot N(g)}{c \cdot n}.$$

It follows that $N(f^{(\ell)}) \ge N^{\ell}(f)/(cn)^{\ell-1}$. We now focus our attention on the case where f is a (nonpartial) function. For this case we can apply known relations between deterministic and nondeterministic communication complexity [1].

CLAIM 5.4. *Let $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$, be a (nonpartial) function. Then, $C(f) \le 2 \log^2 N(f)$.*

Using Theorem 5.1 and Claim 5.4 we get the desired lower bound.

COROLLARY 5.5. *Let $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$, be a (nonpartial) function. Then,*

$$C(f) \ge \overline{C}(f) \ge \sqrt{C(f)/2} - \log n - O(1).$$

Proof. Clearly, $C(f) \ge \overline{C}(f)$. For the other inequality we write

$$\begin{aligned} C(f^{(\ell)}) &\ge \log N(f^{(\ell)}) \\ &\ge \ell \log N(f) - \ell \log n - O(\ell) \\ &\ge \ell \cdot \left(\sqrt{C(f)/2} - \log n - O(1) \right). \end{aligned}$$

By the definition of $\overline{C}(f)$ the result follows. □

We do not know how to extend the above result to general relations or even to partial functions. Our proof method fails in these cases since the gap between deterministic and nondeterministic complexity may be exponential (examples of such partial functions can be constructed based on results is [20]).

6. A function with low amortized randomized complexity. In this section we consider amortized *randomized* communication complexity. Clearly, for every relation f, $\overline{C}_R(f) \leq \overline{C}(f) \leq n$. However, unlike the deterministic case, we do not know whether $\overline{C}_R(f) \leq C_R(f)$ for all relations f. If f is a (partial) function then $\frac{1}{\ell} \cdot C_R(f^{(\ell)})$ is $O(C_R(f) \cdot \log \ell)$ since we can compute f separately for each instance. We do this $O(\log \ell)$ times and take the majority as the output (the $O(\log \ell)$ factor seems to be needed since we require the protocols for computing $f^{(\ell)}$ to be correct with high probability on *all* ℓ instances simultaneously). For specific relations we can do much better. We consider the *identity* function $ID(x, y)$. It is known that $C_R(ID) = \Theta(\log n)$ (see [24]). We show that the amortized complexity of ID with respect to randomized protocols is $\overline{C}_R(ID) = O(1)$. Moreover, the probability of error in our protocol for ID is much less than a constant: it decreases exponentially with $\sqrt{\ell}$. (This can actually be improved to exponential in ℓ.)

To simplify the presentation of the protocols we first assume that the two parties have a way of agreeing on a random string with no cost in communication. This can be thought of as protocols in the *public-coins* model. After presenting the protocols we describe how the parties can agree on such strings while preserving both the communication complexity and the correctness of the protocols.

The following protocol computes the identity function on a *single* pair of inputs (x, y):

- The parties agree on a random string $b \in \{0, 1\}^n$.
- P_1 computes $\langle b, x \rangle$, the inner product of b and $x \pmod 2$, and P_2 computes $\langle b, y \rangle$.
- The parties exchange the bits $\langle b, x \rangle$ and $\langle b, y \rangle$. If the bits are equal they output "equal" ($x = y$), otherwise they output "not equal" ($x \neq y$).

The number of bits exchanged in the protocol is $O(1)$. If $x = y$ it is always correct, while if $x \neq y$ it is correct with probability $\frac{1}{2}$ (which can be improved to any other constant advantage while preserving the $O(1)$ complexity).

Now suppose that the two parties P_1 and P_2 wish to compute the identity function on ℓ input pairs $(x_1, y_1), (x_2, y_2), \ldots, (x_\ell, y_\ell)$. Consider the protocol where P_1 and P_2 amortize the first step in the above protocol while exchanging the bits $\langle b, x_i \rangle$ and $\langle b, y_i \rangle$ for all $1 \leq i \leq \ell$. Such a protocol gives a "good" success probability for computing each of the $f(x_i, y_i)$ separately, while what we want is a "good" probability of computing f on all ℓ instances simultaneously. A possible idea is to decrease the error probability on each (x_i, y_i) to $\frac{1}{poly(\ell)}$ by choosing $k = O(\log \ell)$ vectors $b_i, \ldots, b_k$. Formally, we have the following protocol.

PROTOCOL *multi_compare*.

1. The parties agree on k random strings $b_1, b_2, \ldots, b_k \in \{0, 1\}^n$.
2. For $i = 1, 2, \ldots, k$:
   (a) P_1 computes $u_i = \langle b_i, x_1 \rangle, \langle b_i, x_2 \rangle, \ldots, \langle b_i, x_\ell \rangle$.
   P_2 computes $v_i = \langle b_i, y_1 \rangle, \langle b_i, y_2 \rangle, \ldots, \langle b_i, y_\ell \rangle$.
   (b) The parties exchange the vectors u_i and v_i (each of them is an ℓ-bit string) using a procedure *exchange*(u_i, v_i).
   (c) For $1 \leq j \leq \ell$, if the jth bits of u_i and v_i are different then the parties P_1 and P_2 replace x_j and y_j (resp.) by $x_j = y_j = 0^n$, where 0^n denotes a string of n zeros. (The motivation for this step will become clear while making the analysis below.)
3. The output for the jth pair (x_j, y_j) is "equal" ($x_j = y_j$) if and only if, for every $1 \leq i \leq k$, the jth bits of u_i and v_i are equal.

The probability that the protocol will err on any pair is at most $\ell 2^{-k}$. The only problem with this protocol is that if $k = O(\log \ell)$, and procedure *exchange* in step 2(b) is implemented in a naive way (i.e., P_1 sends u_i to P_2 and P_2 sends v_i to P_1), then the communication complexity of the protocol is $O(\ell \log \ell)$ (i.e., $O(\log \ell)$ invocations of the procedure *exchange*; each requires $O(\ell)$ bits). This complexity is more than our goal.

The main idea for reducing the communication complexity is the following: even if a vector b_i does not recognize all the pairs such that $x_j \neq y_j$, we expect that it does recognize a constant fraction of them. At each time that the parties recognize such a pair, they replace it by $x_j = y_j = 0^n$ (step 2(c)). Therefore, the expected Hamming distance between the vectors u_i and v_i in the above protocol decreases from round to round. We present an implementation of the procedure *exchange*(u, v) that uses this property. It enables the parties to exchange u_i and v_i (step 2(b)) in a cost that depends on the Hamming distance between the vectors; namely, the smaller the Hamming distance, the lower the communication complexity. This will give us the desired complexity.

We start with a simple case where the parties P_1 and P_2 receive, in addition to the input vectors $u, v \in \{0, 1\}^\ell$, respectively, a bound d such that u and v are promised to be at Hamming distance at most d. The following *deterministic* protocol *exchange*$_d(u, v)$ enables each party to learn the value of the other party by exchanging $O(\log \binom{\ell}{d})$ bits (we assume that $d \leq \frac{\ell}{4}$, otherwise the parties simply exchange their inputs). This protocol was discovered independently by Brandman, El-Gamal, and Orlitsky (in [16]), Witsenhausen and Wyner [23], and Karchmer and Wigderson [9].

PROTOCOL *exchange*$_d(u, v)$.

- The parties consider the graph with 2^ℓ nodes corresponding to the strings in $\{0, 1\}^\ell$, and edges between nodes which are at Hamming distance at most $2d$. The parties fix a coloring of the graph. (An effective coloring can be constructed using linear error correcting codes such as BCH.)
- P_1 sends P_2 the color of u and P_2 sends the color of v under the coloring. Since the Hamming distance between u and v is bounded by d and there is at most one member of every color class at distance d from v (as we have a legal coloring of vectors with Hamming distance $\leq 2d$), P_2 can identify u. Similarly, P_1 can identify v.

The degree of every node in this graph is less than $2d \cdot \binom{\ell}{2d}$. Therefore, there exists a coloring of the graph with that many colors. Since the communication in this protocol consists of names of colors, $O(\log(2d \cdot \binom{\ell}{2d})) = O(\log \binom{\ell}{d})$ bits are communicated.

The protocol *exchange*$_d$ above assumes that we have an upper bound on the Hamming distance between u and v. In our case (step 2(b) of the protocol *multi_compare*) a good bound on the distance between u_i and v_i is not known. If we use the protocol *exchange*$_d$ with the wrong bound d then it may fail. Therefore, we generalize the protocol *exchange*$_d$ to a (randomized) protocol *exchange* (which in fact uses *exchange*$_d$ as a procedure). This generalized protocol can work in the case in which a good bound d is not known. The expected number of bits exchanged is still only $O(\log \binom{\ell}{\Delta})$ bits, where Δ is the *actual* Hamming distance between u and v. We use this protocol to implement step 2(b) of the protocol *multi_compare*.

PROTOCOL *exchange*(u, v).

1. The parties agree on k random "test strings" $c_1, c_2, \ldots, c_k \in \{0, 1\}^\ell$.
2. For $d = 2^1, 2^2, 2^3, \ldots, 2^{\log \ell}$:
   (a) P_1 and P_2 engage in *exchange*$_d(u, v)$. Denote the output of P_1 by v' and the output of P_2 by u'.
   (b) *Test step:* P_1 and P_2 test whether $u' = u$ by comparing the inner product of the "test strings" $c_1, c_2, \ldots c_k$ with u and u'; this is done in a bit by bit manner,

quitting early if they discover an error and going to the next d. If all the k bits are equal the protocol terminates (i.e., the parties assume that d is correct, and therefore $u' = u$ and $v' = v$).

By the analysis of the protocol *exchange*$_d$ made above, the number of bits required in step 2(a) of protocol *exchange*(u, v) is $O(\log \binom{\ell}{d})$ if $d \leq \frac{\ell}{4}$ and $O(\ell)$ otherwise. If $u' \neq u$ then the *expected* number of bits exchanged in the test step is $O(1)$. If $u' = u$ then the number of bits exchanged in the test step is $O(k)$; however, this happens only once (note that once we reach d such that $d \geq \Delta$, the deterministic subprotocol *exchange*$_d$ (step 2(a)) always stops with the correct values). Therefore, the expected number of bits communicated is $O(k + \sum_{i=1}^{\log \Delta} \log \binom{\ell}{2^i})$. To compute the overall number of bits communicated we need the following technical claim.

CLAIM 6.1. *For any* $D \leq \ell/2$,

$$\sum_{i=1}^{\log D} \log \binom{\ell}{2^i} = O\left(\log \binom{\ell}{D}\right).$$

Proof. We first show that for all $1 \leq k \leq \frac{\ell}{8}$ we have $\binom{\ell}{2k} \geq \binom{\ell}{k}^{3/2}$. We know that

$$\frac{\binom{\ell}{2k}}{\binom{\ell}{k}} = \frac{(\ell-k)(\ell-k-1)\cdots(\ell-2k+1)}{2k(2k-1)\cdots(k+1)}$$

$$\geq \frac{\ell(\ell-1)\cdots(\ell-k+1)}{2k(2k-1)\cdots(k+1)2^k}$$

$$\geq \frac{\ell(\ell-1)\cdots(\ell-k+1)}{k! \cdot 2^k \cdot 2^k} = \frac{\binom{\ell}{k}}{4^k}.$$

In addition, we have that $\binom{\ell}{k} \geq (\frac{\ell}{k})^k \geq 8^k$ (for the last inequality we use the assumption $k \leq \frac{\ell}{8}$) and hence $\binom{\ell}{2k} \geq \binom{\ell}{k}^2/4^k \geq \binom{\ell}{k}^{3/2}$. Therefore, every term (except perhaps the last two) in the sum $\sum_{i=1}^{\log D} \log \binom{\ell}{2^i}$ is at least at $\frac{3}{2}$ times the preceding term, and the sum is bounded by some constant times the largest term, which is $\log \binom{\ell}{D}$. □

Therefore, the expected number of bits exchanged is $O(k + \log \binom{\ell}{\Delta})$ if $\Delta \leq \frac{\ell}{2}$ and $O(k + \log \binom{\ell}{\ell/2})$ otherwise. The error probability in each round is bounded by 2^{-k} and, therefore, the total error probability is bounded by $\log \ell \cdot 2^{-k}$.

As mentioned, we now use the procedure *exchange* described above to implement step 2(b) of the protocol *multi_compare*. The analysis of the protocol *multi_compare* is as follows: Let D_i be the random variable counting the number of indices $1 \leq j \leq \ell$ such that $\langle b_i, x_j \rangle \neq \langle b_i, y_j \rangle$ but $\langle b_1, x_j \rangle = \langle b_1, y_j \rangle, \ldots, \langle b_{i-1}, x_j \rangle = \langle b_{i-1}, y_j \rangle$. In other words, D_i is the distance between u_i and v_i (recall that if, for some $i' < i$, $\langle b_{i'}, x_j \rangle \neq \langle b_{i'}, y_j \rangle$ and *exchange* succeeds in round i', then both x_j and y_j are replaced by 0^n and, therefore, the jth coordinate of u_i and v_i must be the same).

The expected number of bits exchanged in an execution of the protocol, given that $D_i = d$, is bounded by $c \cdot \sum_{i=1}^{k} (k + \log \binom{\ell}{d})$ for some constant c. For any set of inputs the expected value of D_{i+1}, given that $D_i = d$ and that procedure *exchange* does not fail, is $\frac{d}{2}$. Therefore, conditioned on the fact that *exchange* does not fail in any round, $E[D_i] \leq \ell \cdot 2^{-i}$, and for all $0 \leq m \leq i$ we have $Prob[D_i > \ell 2^{m-i}] < 2^{-m}$. If *exchange* does fail at some round, then just because $D_i \leq \ell$, the expected number of bits exchanged in the protocol is $k \cdot c \cdot (k + \ell)$. The

expected total number of bits exchanged is, therefore, at most

$$E\left[c\cdot\sum_{i=1}^{k}\left(k+\log\binom{\ell}{D_i}\right)\,|\text{exchange always succeeds}\right]$$
$$+\quad Prob\left[\text{exchange fails at least once}\right]\cdot ck\ell$$
$$\le c\cdot k^2+c\cdot\sum_{i=1}^{k}E\left[\log\binom{\ell}{D_i}|\text{exchange always succeeds}\right]+ck\cdot\ell\log\ell 2^{-k}$$
$$\le c\cdot k^2+c\sum_{i=1}^{k}\sum_{m=0}^{i-1}2^{-m}\log\binom{\ell}{\ell 2^{m-i}}+ck\cdot\ell\log\ell 2^{-k}$$
$$\le c\cdot k^2+c\sum_{s=1}^{k}\log\binom{\ell}{\ell 2^{-s}}\sum_{t=0}^{k}2^{-t}+ck\cdot\ell\log\ell 2^{-k}$$
$$\le c\cdot k^2+2c\sum_{s=1}^{k}\log\binom{\ell}{\ell 2^{-s}}+ck\cdot\ell\log\ell 2^{-k},$$

which by Claim 6.1 is $O(k^2+\ell+k\cdot\ell\log\ell 2^{-k})$. If k is $\Theta(\sqrt{\ell})$, then the expected number of bits communicated is $O(\ell)$.

As for correctness, if $x_j\neq y_j$ then with probability at most 2^{-k} we have that for all $1\le i\le k$, $\langle b_i,x_j\rangle=\langle b_i,y_j\rangle$. Therefore, the probability that for some j and all $1\le i\le k$ we have that $\langle b_i,x_j\rangle=\langle b_i,y_j\rangle$ is bounded by $\ell/2^k$. In addition, there is the probability of failure each time we invoke *exchange*(u,v). This probability is at most $\log\ell/2^k$. Thus, the probability of error in our protocol is bounded by $(\ell+k\log\ell)/2^k$. Therefore, if $k=\sqrt{\ell}$ then the probability of error is at most $2^{-\Omega(\sqrt{\ell})}$. To summarize, we have just proved the following lemma.

LEMMA 6.2. *The protocol described above computes, in the public coins model, the identity function on ℓ instances while maintaining that the number of bits communicated is $O(\ell)$ and the probability of error on any instance is at most $2^{-\Omega(\sqrt{\ell})}$.*

Newman [15] has considered the public coins model vs. the private coins model. He showed that $C_R(f)=O(C_{\text{pub}}(f)+\log n)$, which in particular implies

$$C_R(f^{(\ell)})=O(C_{\text{pub}}(f^{(\ell)})+\log\ell n).$$

Clearly,

$$C_R(f^{(\ell)})\ge C_{\text{pub}}(f^{(\ell)})=O(\ell\cdot\log\ell\cdot C_{\text{pub}}(f)).$$

Altogether we have the following theorem.

THEOREM 6.3. *Let $f:\{0,1\}^n\times\{0,1\}^n\to\{0,1\}$ be a (partial) function. Then*

1. $\overline{C}_R(f)=\Theta(\overline{C_{\text{pub}}}(f))$;
2. *for every sufficiently large ℓ, $\frac{1}{\ell}\cdot C_R(f^{(\ell)})=O(\log\ell\cdot C_{\text{pub}}(f))$.*

In particular, this theorem together with Lemma 6.2 gives the following theorem.

THEOREM 6.4. $\overline{C}_R(ID)=O(1)$.

Note, however, that Newman's method is nonconstructive in nature. In the rest of this section we turn to the question of constructively converting the protocols described above to run in the private coins model. We describe a way for the parties to agree on the random strings (i.e., the b_i's and c_i's) with not much additional cost in the communication.

We first describe how to agree on a single string b_i. A collection of vectors $B_m \subset \{0,1\}^m$ is called *ε-biased* if every $x \in \{0,1\}^m$ satisfies $Pr_{b \in B_m}(\langle b, x\rangle = 0) = \frac{1}{2} \pm \varepsilon$. In [14] and [3] the existence and construction of such sets, which are of size polynomial in m (and thus each of their elements can be represented by $O(\log m)$ bits), is shown. For our purposes it is sufficient to take ε to be, say, $\frac{1}{4}$. Fix B_n and B_ℓ, two ε-biased probability spaces. P_1 selects $b_i \in B_n$ by choosing $\log |B_n|$ random bits and sends those bits to P_2. They can both compute b_i. Clearly, if $x = y$ then $\langle b_i, x\rangle = \langle b_i, y\rangle$, while if $x \neq y$ then $\langle b_i, x\rangle \neq \langle b_i, y\rangle$ with probability at least $\frac{1}{4}$. In order to pick k strings $b_1, b_2, \ldots, b_k$ the party P_1 samples k times B_n using $O(k \cdot \log n)$ bits altogether. He sends those bits to P_2. The probability that *multi_compare* errs is at most $\ell \cdot (\frac{3}{4})^k$, and the expectation of D_i is at most $\ell \cdot (\frac{3}{4})^{-i}$.

The strings $c_1, c_2, \ldots, c_k$ are selected similarly from B_ℓ using $O(k \log \ell)$ bits. Note, however, that step (1) in protocol *exchange*(u, v) should not be repeated, i.e., $c_1, c_2, \ldots c_k$ are chosen once and for all at the beginning of the protocol *multi_compare*. In the *public* coins model there is no reason for doing that; we can allow the parties to use new strings $c_1, \ldots, c_k$ each time that step 2(b) of *exchange* is executed. However, the fixed choice of $c_1, \ldots, c_k$ makes the conversion to the *private* coins model easier. Choosing the c_i's once and for all using ε-biased spaces has the property that in protocol *exchange*(u, v) in case $u' \neq u$, the expected number of bits exchanged is $O(1)$. Also, the probability of error is at most $(\frac{3}{4})^k$. Thus the analysis of Lemma 6.2 still applies and we get that the probability of error is at most $2^{-\Omega(\sqrt{\ell})}$ and the number of bits exchanged is $O(\ell + \sqrt{\ell} \log n)$.

For values of ℓ which are around $\log n$ we would like to replace the term $\sqrt{\ell} \log n$ with $\sqrt{\ell} + \log n$. This can be done by sampling the b_i's via a random walk in an expander as in Ajtai, Komlòs, and Szemerèdi [2] (in such a case the b_i's are not independent): The elements of B_n are mapped to nodes of a constant degree expander G. Then, a random walk of length k in G is generated, and the vectors $b_1, b_2, \ldots, b_k$ are the vectors corresponding to the nodes of the walk. The number of bits required to specify the walk is $O(\log |B_n| + k)$, which is $O(\log n + k)$. (See, e.g., [14] for details.) As before, P_1 selects the random bits and sends them to P_2, so that they both agree on the same sequence. If $x \neq y$ then the probability that $\langle b_i, x\rangle = \langle b_i, y\rangle$ for all $1 \leq i \leq k$ goes down exponentially in k. The strings $c_1, c_2, \ldots, c_k$ are selected similarly in B_ℓ using $O(k + \log \ell)$ bits.

To conclude, we have a randomized protocol in the *private* coins model for computing the identity function on ℓ instances with probability of error at most $2^{-\Omega(\sqrt{\ell})}$ and expected complexity of $O(\ell + \log n)$, which is $O(\ell)$ for ℓ sufficiently large. With a "small" additional error the protocol can be converted to a protocol that uses $O(\ell)$ bits in the *worst case*. This gives a constructive proof for Theorem 6.4.

7. Open problems. We conclude by mentioning some open problems:

- In [7] it was conjectured that for any relation f, the communication complexity $\overline{C}(f)$ cannot be smaller than $C(f)$ by more than an additive factor of $O(\log n)$. The examples given in our paper do not contradict this conjecture. On the other hand, according to the best lower bound we are able to prove (Corollary 5.5), even for (nonpartial) functions, that a quadratic gap between $C(f)$ and $\overline{C}(f)$ is possible (and the gap may be even bigger for general relations). Therefore, the main open problem is to try to close this gap by either improving the lower bound (in particular, trying to extend it to relations) or presenting relations with more than $O(\log n)$ difference between $C(f)$ and $\overline{C}(f)$. (Presenting other relations with $O(\log n)$ difference between $C(f)$ and $\overline{C}(f)$ may also be interesting.)
- Another open problem involves trying to achieve similar lower bounds for the *randomized* model. That is, can one prove a lower bound on $\overline{C}_R(f)$ in terms of $C_R(f)$? In the randomized case it is also not known whether $\overline{C}_R(f) \leq C_R(f)$ for every relation f.

- In the case of partial functions f, one can consider a weaker definition for the correctness of a protocol for computing $f^{(\ell)}$: the protocol is required to succeed in computing $f^{(\ell)}(\vec{x}, \vec{y})$ only if, for all i $(1 \leq i \leq \ell)$, we have $|f(x_i, y_i)| = 1$ (otherwise, there is no requirement). In such a model we think of inputs such that $f(x_i, y_i) = \mathcal{D}$ as "illegal." Clearly, proving upper bounds under this definition is easier, while proving lower bounds is harder.

Acknowledgments. We thank Mauricio Karchmer and Avi Wigderson for raising the question and for helpful discussions, and Amos Beimel, Benny Chor, Alon Orlitsky, and Steve Ponzio for many interesting comments on earlier versions of this paper. Finally, we thank an anonymous referee for his very helpful comments and criticism.

REFERENCES

[1] A. Aho, J. Ullman, and M. Yannakakis, *Notions of information transfer in VLSI circuits*, Proc. 15th ACM Symposium on Theory of Computing, ACM Press, 1983, pp. 133–139.

[2] M. Ajtai, J. Komlòs, and E. Szemerèdi, *Deterministic simulation in LOGSPACE*, Proc. 19th ACM Symposium on Theory of Computing, ACM Press, 1987, pp. 132–140.

[3] N. Alon, O. Goldreich, J. Håstad, and R. Peralta, *Simple construction of almost k-wise independent random variables*, Proc. 31st IEEE Symposium on Foundations of Computer Science, St. Louis, MO, IEEE Computer Society Press, 1990, pp. 544–553.

[4] N. H. Bshouty, *On the extended direct sum conjecture*, Proc. 21st ACM Symposium on Theory of Computing, Seattle, WA, ACM Press, 1989, pp. 177–185.

[5] M. Fredman, J. Komlòs, and E. Szemerèdi, *Storing a sparse table with $O(1)$ access time*, J. Assoc. Comput. Mach., 31 (1984), pp. 538–544.

[6] G. Galibati and M. J. Fischer, *On The Complexity of 2-Output Boolean Networks*, Theoret. Comput. Sci., 16 (1981), pp. 177–185.

[7] M. Karchmer, R. Raz, and A. Wigderson, *On proving super-logarithmic depth lower bounds via the direct sum in communication complexity*, Proc. 6th IEEE Structure in Complexity Theory, IEEE Computer Society Press, 1991, pp. 299–304.

[8] M. Karchmer and A. Wigderson, *Monotone circuits for connectivity require super-logarithmic depth*, Proc. 20th ACM Symposium on Theory of Computing, ACM Press, 1988, pp. 539–550.

[9] ———, private communication.

[10] M. Karchmer, E. Kushilevitz, and N. Nisan, *Fractional covers and communication complexity*, SIAM J. Discrete Math., 8 (1995), pp. 76–92.

[11] N. Linial and U. Vazirani, *Graph products and chromatic numbers*, Proc. 30th IEEE Symposium on Foundations of Computer Science, 1989, Research Triangle Park, NC, IEEE Computer Society Press, pp. 124–128.

[12] L. Lovász, *Communication complexity: A survey*, in Paths, Flows, and VLSI Layout, B. H. Korte, ed., Springer-Verlag, Berlin, New York, 1990.

[13] K. Mehlhorn and E. Schmidt, *Las Vegas is better than determinism in VLSI and distributed computing*, Proc. 14th ACM Symposium on Theory of Computing, ACM Press, 1982, pp. 330–337.

[14] J. Naor and M. Naor, *Small-bias probability spaces: Efficient constructions and applications*, SIAM J. Comput., 22 (1993), pp. 838–856.

[15] I. Newman, *Private vs. common random bits in communication complexity*, Inform. Process. Lett., 39 (1991), pp. 67–71.

[16] A. Orlitsky, *Communication Issues in Distributed Communication*, Ph.D. thesis, Stanford University, Stanford, CA, 1986.

[17] ———, *Worst-case interactive communication I: Two messages are almost optimal*, IEEE Trans. Inform. Theory, 36 (1990), pp. 1111–1126.

[18] W. Paul, *Realizing Boolean function on disjoint sets of variables*, Theoret. Comput. Sci., 2 (1976), pp. 383–396.

[19] R. Raz and A. Wigderson, *Monotone circuits for matching require linear depth*, J. Assoc. Comput. Mach., 39 (1992), pp. 736–744.

[20] A. Razborov, *Applications of matrix methods to the theory of lower bounds in communication complexity*, Combinatorica, 10 (1990), pp. 81–93.

[21] Q. F. STOUT, *Meshes with multiple buses*, Proc. 27th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1986, pp. 264–273.

[22] H. S. WITSENHAUSEN, *The zero-error side information problem and chromatic numbers*, IEEE Trans. Inform. Theory, 22 (1976), pp. 592–593.

[23] H. S. WITSENHAUSEN AND A. D. WYNER, *Interframe Coder for Video Signals*, United States Patent number 4, 191, 970, 1980.

[24] A. C. YAO, *Some complexity questions related to distributed computing*, Proc. 11th ACM Symposium on Theory of Computing, ACM Press, 1979, pp. 209–213.

SIAM J. COMPUT.
Vol. 24, No. 4, pp. 751–764, August 1995

005

AN OPTIMAL EXECUTION TIME ESTIMATE OF STATIC VERSUS DYNAMIC ALLOCATION IN MULTIPROCESSOR SYSTEMS*

HÅKAN LENNERSTAD† AND LARS LUNDBERG‡

Abstract. Consider a multiprocessor with k identical processors, executing parallel programs consisting of n processes.

Let $T_s(P)$ and $T_d(P)$ denote the execution times for the program P with optimal static and dynamic allocations, respectively, i.e., allocations giving minimal execution time.

We derive a general and explicit formula for the following maximal execution time ratio: $g(n,k) = \max T_s(P)/T_d(P)$, where the maximum is taken over all programs P consisting of n processes. Any interprocess dependency structure for the programs P is allowed only by avoiding deadlock. Overhead for synchronization and reallocation is neglected.

Basic properties of the function $g(n,k)$ are established, from which we obtain a global description of the function. Plots of $g(n,k)$ are included.

The results are obtained by investigating a mathematical formulation. The mathematical tools involved are essentially tools of elementary combinatorics. The formula is a combinatorial function applied on certain extremal matrices corresponding to extremal programs. It is mathematically complicated but rapidly computed for reasonable n and k, in contrast to the np-completeness of the problems of finding optimal allocations.

Key words. dynamic allocation, static allocation, combinational functions, 0, 1-matrices, extremal combinatorics

AMS subject classifications. 05A05, 05A10, 05A18, 05A20, 05C35

1. Introduction.

In this report an optimal bound of the efficiency of using static compared to dynamic allocation in parallel computing is derived. In static allocation no process reallocation is allowed from the processor where a process was initiated. Dynamic allocation allows unlimited reallocation.

Consider a multiprocessor with k processors. We calculate the quotient

$$g(n,k) = \max_P \frac{T_s(P)}{T_d(P)},$$

where the maximum is taken over all programs P with n processes. The processes of the programs P may have any set of run times and any possible structure of interprocess dependency. $T_s(P)$ and $T_d(P)$ are the execution times for the program P with optimal static and optimal dynamic allocations, respectively, i.e., allocations for which the execution time is minimal. Besides the allocations, the function $g(n,k)$ itself is optimal since it is a bound of the above ratio, which cannot be improved; the term "optimal" thus appears in two senses. The problem is fully defined in §2.

The mathematical aspects of the subject are focused in this report. The same problem is treated from a computer science point of view in [6]. That report also contains a more detailed presentation on how the problem occurs in parallel computing and the significance of the problem for parallel processing.

Since the problems of finding optimal allocations are np-complete, general and, preferably, optimal, results are already needed for a medium number of processors and processes in order to choose multiprocessor architecture which optimizes performance. There appear to be no general results concerning allocation strategies of parallel programs other than the results by Graham [3]. The overhead for process reallocation and synchronization is neglected in that

*Received by the editors November 30, 1992; accepted for publication (in revised form) February 25, 1994. Both authors were supported in part by the Blekinge Research Foundation.

†Department of Telecommunications and Mathematics, University of Karlskrona/Ronneby, S-371 79 Karlskrona, Sweden.

‡Department of Computer Science and Economy, University of Karlskrona/Ronneby, S-372 25 Ronneby, Sweden.

work also. In [3] it is proved that in the unsynchronized case, i.e., when there are no interprocess dependencies, the optimal worst-case ratio of static versus dynamic allocations is 2, taken over all parallel programs consisting of any number of processes. So-called selfscheduling algorithms are also considered. This term is used for dynamic allocation algorithms where, when a processor becomes idle and there are waiting executable processes, one of these is immediately allocated to this processor. It is established in [3] that the execution time for a program allocated with a selfscheduling algorithm is never higher than two times the completion time with optimal dynamic allocation.

The allocation scheme is an important feature for the performance of a multiprocessor, thus an immediate area of applications for the present results is multiprocessor design. The results can further be used to evaluate the efficiency of allocation algorithms versus the best possible algorithm. By using the estimates for the allocation obtained by a selfscheduling algorithm versus an optimal dynamic allocation mentioned above [3], the np-completeness in finding the optimal dynamic allocation can be avoided, and an estimate of the execution time with optimal static allocation is obtained. This is optimal except for at most a factor of 2. The estimate can be improved by running several different selfscheduling algorithms. So-called free-fly algorithms, which allocate parallel programs that are not completely known at start, can also be evaluated in this way in the common case when the complete program is known at completion. The case of average ratio between the allocation strategies is not considered here. We believe that this is a problem which requires a different approach. A consequence of the generality of the present results is that no specific knowledge about a parallel program can be used to improve the estimate. However, it is expected that the techniques presented here can be extended to take advantage of various kinds of program specifics, thus improving the bounds by keeping away from those programs which maximize the ratio studied in this work. In effect we present a methodology for obtaining optimal control of np-complete scheduling problems.

In the mathematical formulation, each static allocation is represented by a partition of the columns of a matrix, which represents a parallel program. The main ingredient in the proof is a duplication argument producing $n!$ copies of the matrix with the columns permuted, which provides control on the optimal static allocation.

Taking the reallocation overhead into account obviously favors static allocation. The significance of this is strongly program dependent, although possibly a feature which can be analyzed by similar mathematical tools.

We next give an overview of the report.

In the following section the allocation problem is described and analyzed in detail. It is formulated as a mathematical problem about so called 0, 1-matrices, i.e., matrices where all entries are 0 or 1. In §3 we give a full formulation of the mathematical problem and introduce necessary notation. Section 4 contains the main result; here the formula for the function $g(n, k)$ is stated and proved. Results which give a global description of the function $g(n, k)$ are presented is §5. Finally, plots describing the function $g(n, k)$ are included.

References [1], [2], [9], and [10] present theoretical results on 0, 1-matrices. References [3], [4], [5], [7], [8], [11], [12], and [13] are a selection of reports where scheduling problems are analyzed. However, none of these, with the exception of [3] as described above, appear to be useful for the present formulation and solution of the problem.

2. The allocation problem. In this context the only difference between static and dynamic allocations is that in the dynamic case processes, after having been started, can be transferred to another processor. The cost of this transference is neglected. In the static case processes are always processed to the end on the processor on which they were initialized. If a process is put into a waiting state, it will be restarted later on the same processor.

A program P consists of n processes of possibly very different execution times. The processes are, of course, usually dependent of each other. One can expect that dependencies of

the type that process i cannot execute further at the time point t_i unless process j has reached the time point t_j. When process j has reached the time point t_j, it is said to execute a synchronizing signal to process i, restarting this process. Certainly there can be many synchronizing signals to a time point t_j, in which case all have to be executed before the process restarts. The execution time of synchronizing signals is neglected. Most parallel programs contain many synchronizing signals. In this report any set of synchronization signals is allowed as long as the parallel program is executable. Our only assumption about the structure of synchronizing signals is that it is consistent, i.e., the program can, for example, when having n processors, execute to the end without violating any synchronizing signal.

Thus a parallel program P of n processes is defined by the n execution times of the n processes and the set of synchronizing signals.

Now consider a parallel program P. Assume that we have found an optimal dynamic allocation with execution time $T_d(P)$. This optimal dynamic allocation will be kept fixed during the entire following argument dealing with the program P and its descendant P'. Next we introduce a discretization of the time interval in subintervals (t_i, t_{i+1}) of equal length such that all synchronizing signals, process initiations, and process terminations appear on the time points t_i, where $t_i = \frac{i}{m} T_d(P), i = 0, \ldots, m$. Obviously, all processes in the interval (t_{i-1}, t_i) are completed before any part of the processes corresponding to the interval (t_i, t_{i+1}) when using this allocation, since this is so without the discretization. Such a discretization is possible if all synchronizing signals and process terminations occur at rational time points, which we can assume. Observe that m might be very large even if the program P is small and has a simple structure. However, m plays no important role in the theory.

Consequently, during a time interval (t_i, t_{i+1}), no process of the program P starts and no process stops.

From the program P we next construct another program P' by two changes of the program P: we introduce new synchronizing signals and prolong certain processes. At every time point t_i we introduce all possible synchronization between the processes. This means that the synchronization now requires that all processes in the interval (t_{i-1}, t_i) have to be completed before any part of the processes corresponding to the interval (t_i, t_{i+1}), which will increase the execution time with most other allocations. Since the execution time of synchronizing signals is neglected, this does not change the total execution time with the fixed optimal dynamic allocation, which is $T_d(P)$. Furthermore, all processors are made to be busy at all time intervals. If necessary, this is achieved by prolonging some processes. However, no process is prolonged beyond $T_d(P)$, hence $T_d(P) = T_d(P')$. It is of no importance that the transformation from P to P' can be made in many ways; many programs can play the role of P' to a specific program P.

By the construction we thus have $T_d(P) = T_d(P')$. However, since introducing more synchronization and prolonging processes never shortens the execution time, the execution time is either increased or unchanged for other allocations. In particular, for optimal static allocation we therefore have $T_s(P) \leq T_s(P')$. Consequently,

$$\frac{T_s(P)}{T_d(P)} \leq \frac{T_s(P')}{T_d(P')}.$$

Certainly there are programs P which are left unchanged by the above transformation: programs such that $P = P'$. Since these programs constitute a subset of the parallel programs we consider, we actually have

$$\max_P \frac{T_s(P)}{T_d(P)} = \max_P \frac{T_s(P')}{T_d(P')}.$$

Therefore, in order to calculate the maximum, only programs of the type P' need to be considered.

We next represent a program P' with optimal dynamic allocation on a multiprocessor with k processors by an $m \times n$ matrix of 0's and 1's only. Here each process is represented by a column and each time period is represented by a row. The entry at the position (i, j) of the matrix is 1 if the jth process is active between t_{i-1} and t_i; if it is inactive the entry is 0. Each row contains exactly k 1's, since each processor is constantly busy. In this report, such a matrix is referred to as an m, n, k-type matrix. The main part of this paper analyzes these matrices. For example, we characterize the type of matrix which corresponds to the worst case. Because of the complete synchronization, each row has to be completed before the next row. The optimal dynamic allocation of the program P' represented in this way is m, i.e., the time unit is changed to $T_d(P)/m$.

What is the optimal static execution time of the program P'? To compute this we need to decide how the n processes are to be allocated to the k processors. Since every process in the static case is to be executed on one processor only, the static allocation corresponds to a way of grouping the n columns of the matrix together in k sets, one set for each processor. Because of the complete synchronization, at each step the processors have to wait for the slowest processor. This is the processor that has the highest number of processes to execute, i.e., the maximum number of 1's in one group. Hence the static execution time is the sum of the maximas for the rows multiplied by the factor $T_d(P)/m$. This is the *optimal* static execution time $T_s(P')$ if we have found the best allocation, i.e., a way of grouping the n columns together in k sets which minimizes the static execution time. In the following we denote

$$T(P) = T_s(P)\frac{m}{T_d(P)},$$

i.e., we compute the time in the time unit $T_d(P)/m$.

In the main result of this paper we give a formula for the function $g(n, k)$ representing the worst case, i.e., for any parallel program P_0,

$$\frac{T_s(P_0)}{T_d(P_0)} \leq g(n, k) = \max_P \frac{T_s(P)}{T_d(P)} = \max_P \frac{T(P)}{m}.$$

Here $T(P)$ is defined in the next section.

3. The mathematical formulation. As described in the previous section, the corresponding mathematical problem can be formulated as follows.

Consider an $m \times n$ matrix P of 0's and 1's only such that each row has exactly k 1's and thus $n - k$ 0's, $1 \leq k \leq n$. These matrices are referred to as m, n, k-type matrices.

The column vectors of P will sometimes be denoted by v_i. Consider a partition A of the n vectors v_i into k sets. Observe that the number of sets equals the number of 1's on each row in P. We will be mostly concerned with partitions where the sizes of the sets in the partition differ as little as possible. If n/k is an integer w then every set has w members. Denote the integer part of n/k, the floor function, by $\lfloor n/k \rfloor$, and the smallest integer greater than or equal to n/k, the ceiling function, by $\lceil n/k \rceil$. If n/k is not an integer, the sets in a partition where the sizes differ as little as possible have $\lfloor n/k \rfloor$ or $\lceil n/k \rceil$ members.

Given any partition A of the column vectors in k sets, we form a quantity $T_A(P)$ as follows.

The vectors in each group are added together, making k vectors of nonnegative integers from the n column vectors. By taking the maximum of these vectors componentwise, one single vector of positive integers is obtained. All vectors here are, of course, m-dimensional. The sum of the components of the final vector is the quantity $T_A(P)$. Set as a formula we

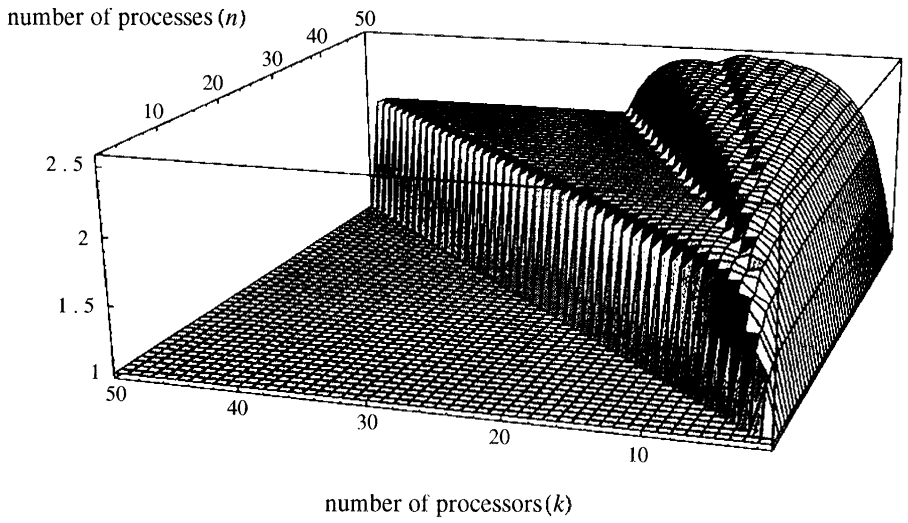


FIG. 1. *The function $g(n, k)$ is defined as the worst-case ratio of static versus dynamic allocations: $g(n, k) = \max_P(T_s(P)/T_d(P))$. Here $T_s(P)$ and $T_d(P)$ denote the execution time for a parallel program P with optimal static and optimal dynamic allocations, respectively, executed on a multiprocessor with k identical processors. The maximum is taken over all parallel programs P consisting of n processes.*

have

$$T_A(P) = \sum_{j=1}^{m} \max_{l=1,\ldots,k} \left(\sum v_i \right),$$

where the last sum is taken over the indices i, which belong to the lth partition set.

We want to choose the partition A so that $T_A(P)$ is minimal. The interesting quantity is thus

$$T(P) = \min_{\text{all partitions } A} T_A(P).$$

The function $g(n, k)$ is defined by

$$g(n, k) = \max \left\{ \frac{T(P)}{m}, \text{all } m, n, k - \text{type matrices } P \right\}.$$

For given m, n, and k, we will thus be concerned with the problem of calculating $\max T(P)/m$ over all m, n, k-type matrices P.

A natural conjecture is the estimate $T(P)/m \leq g(n, k) \leq 2$. In the case $n \leq 2k$ it is immediately seen to be true by simply grouping the column vectors together in pairs. The conjecture is not true in the general case as is immediately visible in Fig. 1. However, for a partition where the size of the sets differ as little as possible, the largest set has $\lceil n/k \rceil$. Then the maximum number of 1's in a set is $\min(\lceil n/k \rceil, k)$: both the number of slots in the largest set and the total number of 1's provide bounds. We thus obtain a crude estimate $g(n, k) \leq \min(\lceil n/k \rceil, k)$. We will frequently compare the optimal estimate $g(n, k)$ with this crude estimate.

Our final preparation before the main result is to introduce and summarize the notation which is relevant in this situation.

We say that a matrix P is of m, n, k-*type* if it has m rows and n columns, all entries are 0's or 1's, and each row has exactly k 1's, where $1 \le k \le n$.

We call a matrix P *complete* if all possible rows, that is, if all $\binom{n}{k}$ permutations of the k 1's, occur equally frequently as rows of P. The number of rows is thus necessarily divisible by $\binom{n}{k}$.

We also need the following three combinatorial functions. Let I be a finite sequence of nonnegative integers. Then we define

$b(I)$ = the number of distinct integers in I;

$a(I, j)$ = the number of occurences of the jth distinct integer in I, enumerated in size order, $1 \le j \le b(I)$;

$\pi(k, w, q, l)$ = the number of permutations of q 1's distributed in kw slots, which are divided in k sets with w slots in each such that the set with maximum number of 1's has exactly l 1's.

4. The optimal estimate. With this notation, the main result is contained in the following theorem.

THEOREM 1. *Given positive integers m, n and k, $k < n$, in the case where $w = n/k$ is an integer, we have for all matrices P of m, n, k type,*

$$T(P)/m \le g(n,k) = \frac{1}{\binom{n}{k}} \sum_{l=1}^{\min(w,k)} l\pi(k, w, k, l).$$

If $w = n/k$ is not an integer, we let $w = \lfloor n/k \rfloor$ and denote the remainder of n divided by k by n_k, i.e., $n_k = n - k\lfloor n/k \rfloor$. Then we have for all matrices P of m, n, k type,

$$T(P)/m \le g(n,k) = \frac{1}{\binom{n}{k}} \sum_{l_1=\max(0,\lceil \frac{k-(k-n_k)w}{n_k} \rceil)}^{\min(w+1,k)} \sum_{l_2=\max(0,\lceil \frac{k-l_1 n_k}{k-n_k} \rceil)}^{\min(w,k-l_1)} \max(l_1, l_2)$$

$$\cdot \left(\sum_{i=\max(l_1, k-l_2(k-n_k))}^{\min(l_1 n_k, k-l_2)} \pi(n_k, w+1, i, l_1)\pi(k-n_k, w, k-i, l_2) \right).$$

The function $\pi(k, w, q, l)$ is 0 if $\min(q, w) < l$ or $q > kl$, otherwise it is given by

$$\pi(k, w, q, l) = \binom{w}{l} \sum_{I} \binom{w}{i_1} \cdot \cdots \cdot \binom{w}{i_{k-1}} \frac{k!}{\prod_{j=1}^{b(\{l, i_1, \ldots, i_{k-1}\})} a(\{l, i_1, \ldots, i_{k-1}\}, j)!}.$$

Here the sum is taken over all sequences of nonnegative integers $I = \{i_1, \ldots, i_{k-1}\}$, which are decreasing, $i_j \ge i_{j+1}$ for all $j = 1, \ldots, k-2$, bounded by l, $i_1 \le l$, and have sum $q - l$, $\sum_{j=1}^{k-1} i_j = q - l$.

For each m, n, k-type matrix P the minimum

$$T(P) = \min_{\text{all partitions } A} T_A(P)$$

is attained for a partition where the sizes of the sets in the partition differ as little as possible.

The bound is optimal in the sense that if $\binom{n}{k}$ divides m, in which case there exist complete matrices, we have $T(P)/m = g(n, k)$ for all complete matrices P.

In Fig. 1 a plot of the function $g(n, k)$ for $1 \le k, n \le 50$ is presented. Figure 2 shows the detailed structure of $g(n, k)$.

The ridge/valley structure of the surface $g(n,k)$ is shown here in detail by studying the function $g(n,k)$ as a function of k, for each constant n. The locations of global maximas (■), local maximas (■, ▩) and local minimas (▦) are plotted.

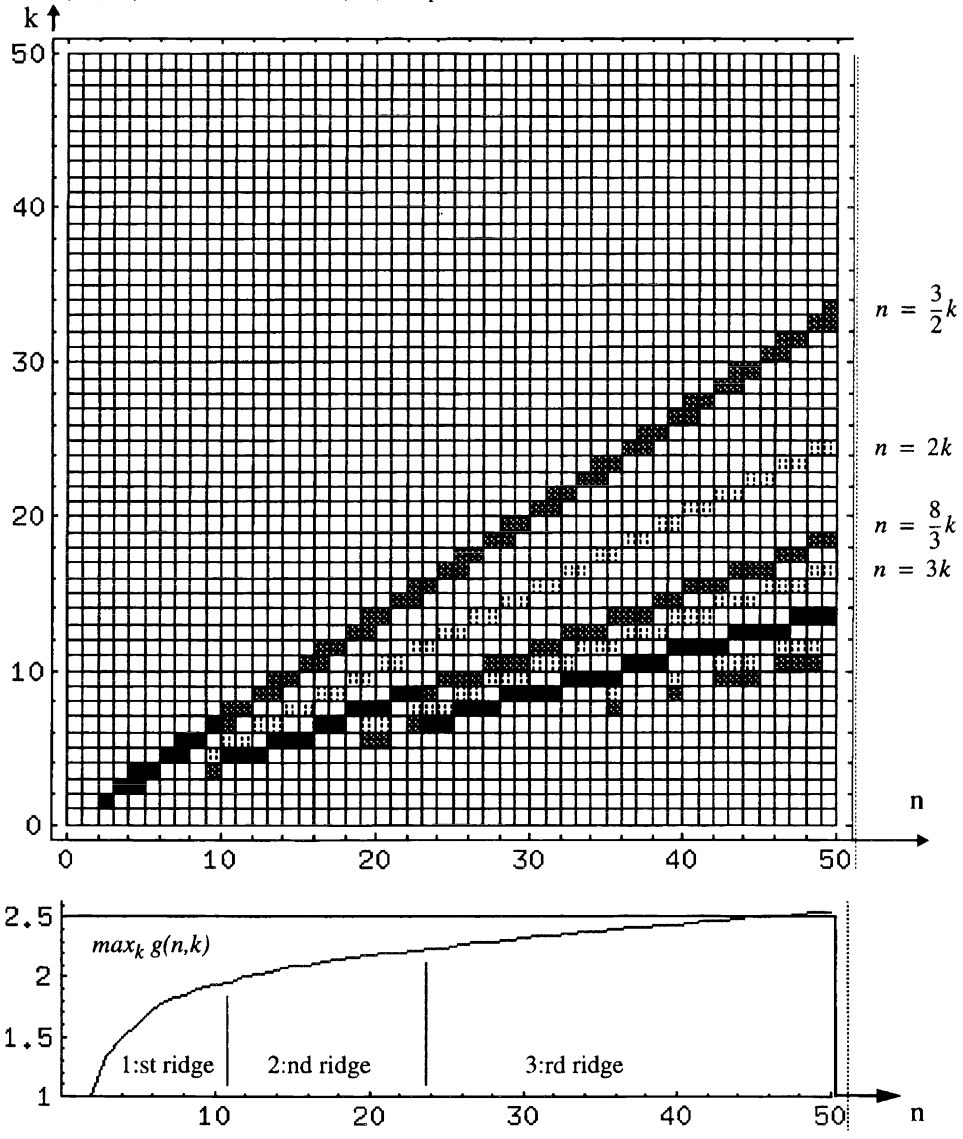


FIG. 2. *Notes:* (1) $g(n, k)$ *is increasing as a function of* n. (2) *The straight line equations in the margin are simply constructed from the graph.* (3) *The function* $\max_k g(n, k)$ *gives the worst case ratio for all parallel programs with n processes and for all multiprocessors with identical processors.*

If $k \geq n$, the definition $g(n, k) = 1$ is consistent with the allocation problem. Then each processor has one single process to execute and there is no difference between static and dynamic allocations. The case $g(n, 1) = 1$ is another trivial case.

The sequences in the function π can easily be generated by the algorithm described in the following lemma. We say that the least decreasing sequence of length μ and sum σ is the sequence $\{\lceil \frac{\sigma}{\mu} \rceil, \ldots, \lceil \frac{\sigma}{\mu} \rceil, \lfloor \frac{\sigma}{\mu} \rfloor, \ldots, \lfloor \frac{\sigma}{\mu} \rfloor\}$. If σ_μ is the remainder when σ is divided by μ, the number of $\lceil \frac{\sigma}{\mu} \rceil$'s is σ_μ and the number of $\lfloor \frac{\sigma}{\mu} \rfloor$'s is $\mu - \sigma_\mu$, making the sum of the sequence σ.

LEMMA 1. *Let λ and σ be nonnegative integers and μ be a positive integer such that $\lambda \le \sigma \le \lambda\mu$.*

Every sequence of μ integers in the interval $0 \le i \le \lambda$, which is decreasing, bounded by λ, and has sum σ, is generated exactly once by the following algorithm:

1. *Take I as the least decreasing sequence of length μ and sum σ.*
2. *Find the* rightmost *position in I, say j_1, which fulfills*
   (a) $i_{j_1} < l$,
   (b) $i_{j_1} < i_{j_1-1}$ *or* $j_1 = 1$,
   (c) $i_{j_1+1} > 0$.
   The algorithm terminates if no such j_1 can be found.
3. *The next sequence is obtained from I by increasing the entry in position j_1 by one and replacing the subsequence $\{i_{j_1+1}, \dots, i_\mu\}$ with the least decreasing subsequence of length $\mu - j_1$ and sum $\sum_{j=j_1+1}^{\mu} i_j - 1$.*
4. *Go to step* 2.

Proof of the lemma. It is immediately clear that the starting sequence is decreasing, has sum σ, and has entries in the interval $0 \le i \le l$. It is also obvious that these properties are preserved by the algorithm.

Consider a sequence I of this kind. If it is not the last one generated by the algorithm, the next sequence will have its entry at j_1 increased. This entry will not decrease again unless an entry to the left is increased. By reapplying this argument we find that no sequence is generated twice by the algorithm.

It remains to prove that the algorithm generates all such sequences. This is done by induction over the length μ of the sequence.

For $\mu = 1$ the only sequence is $\{\sigma\}$, which is the starting sequence of the algorithm and the only sequence generated by the algorithm.

Assume that the lemma is true for all decreasing sequences of length μ.

We want to prove that the algorithm generates all decreasing sequences of length $\mu + 1$, sum σ, and bound λ.

These sequences are $\{\min(\lambda, \sigma), I_1\}, \dots, \{\lceil \sigma/\mu \rceil, I_{j_0}\}$, where $I_1, \dots, I_{j_0}$ are decreasing sequences of length μ bounded by $\min(\lambda, \sigma), \dots, \lceil \sigma/\mu \rceil$ and with sums $\sigma - \min(\lambda, \sigma), \dots, \sigma - \lceil \sigma/\mu \rceil$, respectively. Now the algorithm applied on the sequences $\{\min(\lambda, \sigma), I_1\}, \dots, \{\lceil \sigma/\mu \rceil, I_{j_0}\}$ of length $\mu + 1$ with the bounds $\min(\lambda, \sigma), \dots, \lceil \sigma/\mu \rceil$, is, except for the first entry, the same as the algorithm applied on the sequences $\{I_1, \dots, I_{j_0}\}$ of length μ with the bounds $\min(\lambda, \sigma), \dots, \lceil \sigma/\mu \rceil$. By the induction assumption this generates all decreasing sequences of length μ. The lemma is proved.

Proof of the theorem. Consider an arbitrary matrix P of m, n, k type. Let A be a partition where the sizes of the sets differ as little as possible. We will later prove that the minimum is attained for such a partition if P is complete. This kind of partition is enough to consider for an arbitrary m, n, k-type matrix P since we clearly have

$$\min_{\text{any partition } A} \frac{T_A(P)}{m} \le \min_{\text{any partition } A \text{ with sets of "equal" size}} \frac{T_A(P)}{m}.$$

In the following we derive the optimal upper bound $g(n, k)$ of the rightmost quantity.

Note that some m, n, k-type matrices do not have an optimal partition of this type. An example of this is the 3, 4, 2-type matrix

$$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}.$$

Each row in P can be regarded as a permutation of k 1's in n slots. There are of course $\binom{n}{k}$ such permutations. Furthermore, there exist $n!$ permutations of the columns of P, where each permutation produces a possibly different m, n, k-type matrix $P_i, i = 1, \ldots, n!$. When we permute the column vectors, we permute the rows; all possible rows appear if we perform all possible column permutations. Now partition A applied to P_i is equivalent to a different partition A_i applied to the original matrix P in the sense that $T_A(P_i) = T_{A_i}(P)$.

Next we construct a matrix P^* from the matrices P_i, which has $m\,n!$ rows and n columns, using the following duplication argument. This will provide control of the partitions.

The first m rows of P^* constitute the matrix P itself. The next m rows constitute the matrix P where the columns are permuted according to a specific permutation, which is not the identity. The following $n! - 2$ blocks of m rows contain all other permutations of the columns of the matrix P. We know three facts about the matrix P^* which make this procedure useful.

1. Every row in P^* occurs exactly as many times as any other different row in the matrix P^*. Every possible row does appear. That is, P^* is a complete matrix.
2. Each column permutation represents a partition A_i of the columns of P, so that $T_A(P_i) = T_{A_i}(P), i = 1, \ldots, n!$.
3. The quantities $T_A(P_i), i = 1, \ldots, k!$ relate to the quantity $T_A(P^*)$ as $T_A(P^*) = \sum_{i=1}^{n!} T_A(P_i)$.

The first fact is clear since the duplication argument from each row produces all other $n! - 1$ rows, counting all 0's and all 1's as different. In reality there are $\binom{n}{k}$ different such rows, hence each is repeated $k!(n-k)!$ times. Fact 2 follows by an argument above. Observe that the function T is by definition the sum of an operation made on each row separately. On the right side of the equality in fact 3 this summation is carried out separately for the $n!$ parts of P^*.

Thus, since $T(P)$ arises from the partition A_i, which gives the smallest value of $T_{A_i}(P) = T_A(P_i)$, we have $T(P) \leq T_{A_i}(P) = T_A(P_i)$ for all $i = 1, \ldots, n!$. We then obtain from fact 3,

$$T(P)/m \leq \sum_{i=1}^{n!} T(P_i)/n!m = T(P^*)/n!m\,.$$

We have thus established that the complete matrices are extremal matrices for the present problem.

A complete matrix has a particularly simple structure. Each permutation in the matrix P^* is repeated m times if we count every 1 and every 0 as distinct. By releasing this distinction, each permutation really is repeated $mn!(n-k)!$ times. Since this factor only multiplies all occurring numbers, what is left to study is the complete matrix $\tilde{P}$, where each permutation occurs exactly once. We have

$$T(P^*) = mn!(n-k)!T(\tilde{P}), \text{ so}$$

$$T(P) \leq T(P^*)/n! = mn!(n-k)!T(\tilde{P})/n! = \frac{m}{\binom{n}{k}} T(\tilde{P}).$$

The matrix $\tilde{P}$ has $\binom{n}{k}$ rows and n columns, and contains each permutation of the k 1's in the n slots exactly once. The columns are grouped together in sets with $\lceil n/k \rceil$ or $\lfloor n/k \rfloor$

members. For the maximum operation we are interested in the number of 1's in the set with most 1's, we have here, say, l 1's. For the component sum of the final vector we are interested in the number of permutations where we have exactly l 1's in the set of maximum 1's.

First we consider the case when $n/k = w$ is an integer. Then all sets in the partition contain w vectors.

For the sake of clarity, we compute a few special cases of $\pi(k, w, k, l)$ before considering the general situation. In these cases we assume that $k \leq w$.

1. Clearly, $\pi(k, w, k, 1) = w^k$, since here we have to put exactly one 1 in each set. In the first set the 1 can be put in w different slots as it can in all other k sets.
2. $\pi(k, w, k, k) = \binom{w}{k} k$. Here the first factor comes from the ways of putting all the 1's in one set, and the factor k is the number of sets where this can be done.
3. $\pi(k, w, k, k-1) = \binom{w}{k-1} w \binom{k}{2}$. The second factor in this case is the number of slots to put the single 1. The last factor is the number of ways to distribute the two different sets among the k sets.
4. $\pi(k, w, k, k-2) = \binom{w}{k-2}(\binom{w}{2}\binom{k}{2} + w\binom{k}{3}/2)$. The first term is the number of ways to produce $l = k-2$ with the distribution $k-2, 2, 0, \ldots, 0$ of the 1's in the k sets. The other term arises from the distribution $k-2, 1, 1, 0, \ldots, 0$.

Generally, to begin we have a number of ways to distribute k 1's in k sets regardless of order both in each set and between the sets. These ways are represented by the decreasing sequences. The order in each set is disregarded here in such a way that only the number of 1's is significant. The order between the sets is disregarded by choosing one specific order, which is decreasing sequences.

In a set of i 1's and $w-i$ 0's there are $\binom{w}{i}$ different permutations. So, taking the order in the sets into account, if we have $\{i_1, \ldots, i_k\}$ 1's in the k sets, respectively, there are $\binom{w}{i_1} \cdot \cdots \cdot \binom{w}{i_k}$ different permutations. Since the maximum number l must appear, we always get a factor $\binom{w}{l}$, which can be factored out. The remaining sequence is of length $k-1$ and has sum $k-l$.

Next we consider the order between the sets. There are $k!$ permutations of the sets. If there are $a(\{l, i_1, \ldots, i_{k-1}\}, j)!$ sets with i_j 1's, we get no new permutations by permuting within this group of sets. Hence the factor from permuting the sets is

$$\frac{k!}{\prod_{j=1}^{b(\{l,i_1,\ldots,i_{k-1}\})} a(\{l, i_1, \ldots, i_{k-1}\}, j)!}.$$

The number of permutations which give the number l is thus

$$\pi(l) = \binom{w}{l} \sum_{I} \binom{w}{i_1} \cdot \cdots \cdot \binom{w}{i_{k-1}} \frac{k!}{\prod_{j=1}^{b(\{l,i_1,\ldots,i_{k-1}\})} a(\{l, i_1, \ldots, i_{k-1}\}, j)!},$$

summing over the decreasing sequences of length $k-1$, sum $k-l$ and bound l.

In the case when n/k is not an integer, we work with a partition where the n_k leftmost sets have $w+1 = \lceil n/k \rceil$ vectors and the rightmost $k-n_k$ have $w = \lfloor n/k \rfloor$ vectors.

The formula in this case is derived from the previous formula by introducing the possibility that the number of sets k and the number of 1's q are not equal. By summing over all possible maximums to the left (l_1) and right (l_2), and over all possible numbers of 1's to the left (i), the results for general n and k follow. The bounds of the indexes appear from the limitations of the number of 1's for which there is room to the left or to the right in the different cases, and from the minimum number of 1's according to l_1, l_2, and i.

We finally prove the optimality result concerning the type of partition.

If the matrix P is complete with $\binom{n}{k}$ rows then $P = \tilde{P}$, and the above calculation is true with equality if $\min_{\text{all partitions } A} T_A(P)$ is attained for a partition where the sizes of the sets differ

as little as possible. Next we prove that this holds for complete matrices. Thus the bound is attained for programs corresponding to complete matrices.

Let A be a partition of the n columns into k sets with $\{i_1, \ldots, i_k\}$ columns in each set, respectively. Assume that there are i_j's, say i_1 and i_2, such that $i_1 \geq i_2 + 2$, otherwise we have the aforementioned type of partition. From the partition A we will obtain a new partition A' by transferring the i_1th vector from the first set to the second. We show that the result is never worse for this partition, i.e., $T_{A'}(P) \leq T_A(P)$. By repeating this transformation a partition where the sizes of the sets differ as little as possible is finally derived, and the result follows from the inequality.

Consider a row in $\tilde{P}$, that is, a permutation $p = \{p(i)\}_{i=1}^{n}$ of 1's and 0's. If $p(i_1) = 0$ nothing happens on this row. If $p(i_1) = 1$ and $\sum_{i=1}^{i_1} \leq \sum_{i=i_1+1}^{i_2}$, the maximum taken over this row does increase unless other sets contain more 1's. These are the critical permutations. However, for each such permutation there is a unique permutation $\tilde{p}$, where the row maximum in such a case will decrease. $\tilde{p}$ is defined as a partial mirror image:

$$\tilde{p}(i) = \begin{cases} p(i_1 + i_2 + 1 - i) & \text{if } i = 1, \ldots, i_2 \text{ or } i = i_1 + 1, \ldots, i_1 + i_2, \\ p(i) & \text{all other } i. \end{cases}$$

Since $\sum_{i=1}^{i_2} p(i) < \sum_{i=i_1+1}^{i_1+i_2} p(i)$ for a critical permutation, we know that $\tilde{p}$ is not critical if p is. Furthermore, it follows from $\tilde{p} = p$ that $\tilde{}$ is a bijection. Hence every critical permutation p can be paired with a unique permutation $\tilde{p}$ since $\tilde{P}$ contains each permutation exactly once.

Because $p(i) = \tilde{p}(i)$ for $i = i_1 + i_2 + 1, \ldots, n$ and $\sum_{i=1}^{i_1} \tilde{p}(i) \geq \sum_{i=i_1+1}^{i_2} p(i)$, it is clear that if the partition change causes the maximum on the row with p to increase, then the maximum on the row with $\tilde{p}$ will certainly decrease. The proof of the theorem is completed.

Given n and k, how many matrices of $\binom{n}{k}, n, k$ type are complete? Since we obtain all complete matrices by permuting the rows in a given complete matrix, there are $\binom{n}{k}!$ complete matrices. There are in total $\binom{n}{k}^{\binom{n}{k}}$ different matrices of $\binom{n}{k}, n, k$ type: for each row there are $\binom{n}{k}$ possibilities and we have $\binom{n}{k}$ rows. The relative number of complete matrices can now be estimated by Stirling's formula $n! \approx (\frac{n}{e})^n \sqrt{2\pi n}$:

$$\frac{\binom{n}{k}!}{\binom{n}{k}^{\binom{n}{k}}} \approx \left(\frac{1}{e}\right)^{\binom{n}{k}} \sqrt{2\pi \binom{n}{k}},$$

which tend to zero rapidly as $\binom{n}{k} \to \infty$. The significance of this is limited, however, since it is clear that for increasing n and k, matrices very close to complete matrices play a role increasingly similar to the role of the complete ones.

5. Properties of the function $g(n, k)$. The function $g(n, k)$ can be regarded as a weighted mean value of the integers $l = 1, 2, \ldots, \min(\lceil n/k \rceil, k)$, where the weights are the number of permutations which gives l to the final sum, divided by the total number of permutations $\binom{n}{k}$. This fact is exploited in this section. The crude estimate can be viewed as the estimate of this weighted mean value by the largest integer $\min(\lceil n/k \rceil, k)$.

From Theorem 2 we will be able to derive the following description of $g(n, k)$ for large n and k. We find that $g(n, k)$, like the crude estimate, has an infinite series of plateaus: for each positive integer w there is an almost planar unbounded part which is a subset of the domain $n > (w - 1)k, n < wk$, where the values are close to w. After the proof of the theorem this property is given a more precise formulation. In Fig. 1 the first two plateaus, $g = 1$ and $g = 2$ can be seen. It seems like the distance from the origin to the plateau $g = z$ increases very rapidly with z. The plateau $g = 3$, for example, appears beyond $n = 100$.

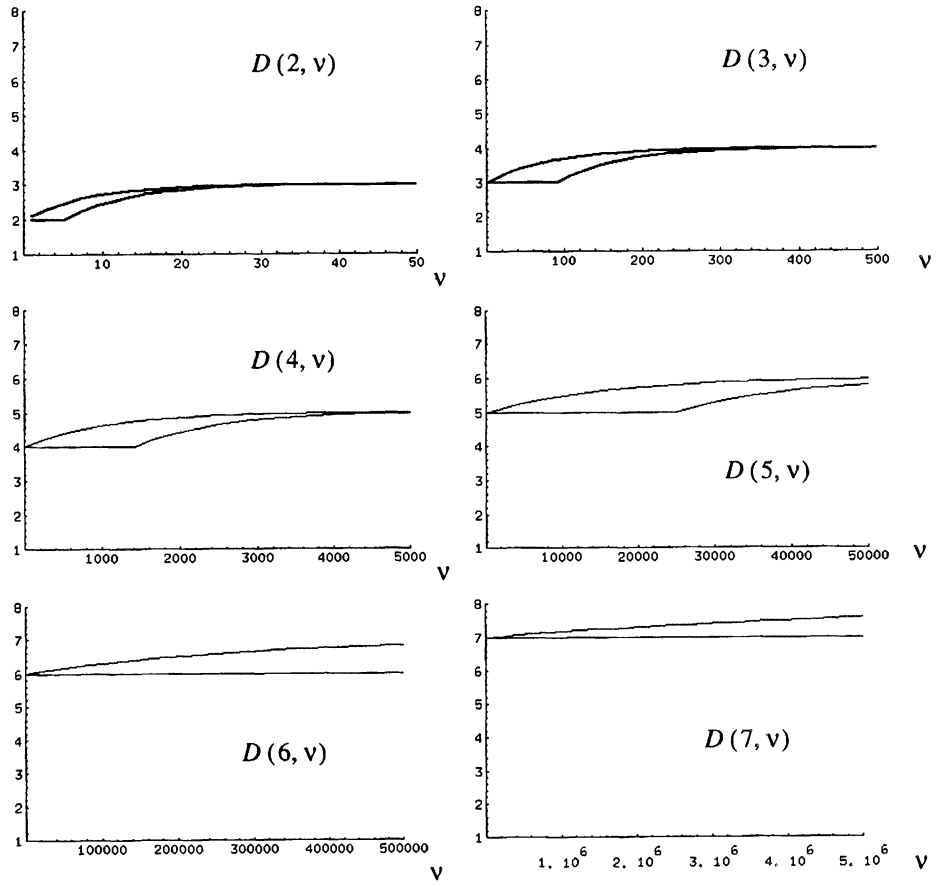


FIG. 3. *Functions giving upper and lower bounds for the diagonal limit functions* $D(w, \nu) = \lim_{k\to\infty} g(wk + \nu, k)$ *are plotted, given by the estimates* $\max(w, w + 1 - w(1 - w^{-w-1})^{\nu}) \leq D(w, \nu) \leq w + 1 - (1 - w^{-w-1})^{\nu}$. *Notes:* (1) *Observe the increasing scale on the* ν*-axis.* (2) *The plots illustrate the fact that, at least for large* k*, the distance from the* w*th plateau to the next increases very rapidly with* w.

The crude and optimal estimates often differ strongly, quantitatively. One simple example of this is the fact that for $1 \leq n \leq 100$, $1 \leq k \leq 100$, the maximum of the crude estimate is 10 while the maximum of g is 2.8.

THEOREM 2. *The function* $g(n, k)$ *has the following properties:*

1. $g(n, k)$ *is increasing as a function of* n.
2. *For any positive integer* w, $g(wk, k)$ *is increasing as a function of* k.
3. *For all positive integers* ν *and* w *we have*

$$w + 1 - (1 - w^{-(w+1)})^{\nu} \geq \lim_{k\to\infty} g(wk + \nu, k) \geq \max(w, w + 1 - w(1 - w^{-(w+1)})^{\nu}).$$

4. *The function* $g(n, k)$ *is unbounded.*

In Fig. 3 the functions bounding $D(w, \nu) = \lim_{k\to\infty} g(wk + \nu, k)$ are plotted for $w = 2, \ldots, 7$.

Proof. (1) $g(n, k) = \min_{\text{all partitions } A} T_A(P)/m$, where P is a complete m, n, k-type matrix. Let A be a partition where the sizes of the sets differ as little as possible. Now, by adding to the matrix P one column of zeros only, we obtain the noncomplete $m, n + 1, k$-type matrix

P_0. This column is added to the partition A, producing A_0, in such a way that A_0 also have sizes of the sets which differ as little as possible. By the proof of Theorem 2 it follows that $g(n, k) = T_A(P)/m = T_{A_0}(P_0)/m \leq T_{A_0}(\tilde{P_0})/m = g(n+1, k)$, where $\tilde{P_0}$ is a complete matrix of $m, n+1, k$ type. We can choose m so that it is divisible by both $\binom{n}{k}$ and $\binom{n+1}{k}$.

(2) The increasing property in property 2 follows by adding w columns to the matrix P with an argument similar to the one in the proof of property 1.

(3) Assume that k is large, so that $k \leq (w+1)v$. When attempting to estimate $g(wk+v, k)$ we of course consider m, $wk+v$, k-type matrices, so we have v sets with $w+1$ column vectors and $k - v$ sets with w vectors. The limit will be derived by estimating the weight to $w+1$.

How many of the total $\binom{wk+v}{k}$ permutations give rise to $w+1$? To begin, $\binom{wk+v-(w+1)}{k-(w+1)}$ of the permutations have all 1's in the first set. There are v sets with size $w+1$, so we have v sets of permutations with at least one set with all 1's. We denote these sets by B_{1j}, $j = 1, \ldots, v$. Each of these sets of permutations has $\binom{wk+v-(w+1)}{k-(w+1)}$ members. The sets are not disjoint, for example, permutations which have all 1's in at least two of the sets appear in two sets of permutations. Analogously, we have $\binom{v}{2}$ sets of permutations, the B_{2j}'s, $j = 1, \ldots, \binom{v}{2}$, which have at least two sets full of 1's. These sets have cardinality $\binom{wk+v-2(w+1)}{k-2(w+1)}$. Observe that each B_{2j} is the intersection of two B_{1j}'s.

In general, there are $\binom{v}{i}$ permutation sets B_{ij}'s having at least i sets of all 1's, each of which has $\binom{wk+v-i(w+1)}{k-i(w+1)}$ members. Each B_{ij} is the intersection of i $B_{1,j}$'s. Now recall that if A_i are n finite sets, we have the relation

$$|\cup_{i=1}^{n} A_i| = \sum_{i=1}^{n} (-1)^{i+1} \left(\sum_{\text{all sets } I \subset \{1,2,\ldots,n\} \text{ with } |I| = i} |\cap_{j \in I} A_j| \right).$$

This generalizes the rule $|A \cup B| = |A| + |B| - |A \cap B|$ for finite sets A and B.

By applying the relation, it follows that the number of permutations which have all 1's in at least one set of size $w+1$ is

$$\sum_{i=1}^{v} (-1)^{i+1} \binom{v}{i} \binom{wk + v - i(w+1)}{k - i(w+1)}.$$

Next we divide by $\binom{wk+v}{k}$ and let $k \to \infty$. What remains of the sum is then

$$\sum_{i=1}^{v} (-1)^{i+1} \binom{v}{i} w^{-i(w+1)} = 1 - (1 - w^{-(w+1)})^v.$$

The main part of the estimates in property 3 now follows from the weighted sum. By letting $v \to \infty$ in the limit applied on $n = (w-1)k + v$, it follows that the limit on $n = wk + v$ is bounded from below by w and property 3 follows.

(4) One consequence of the theorem is that g is bounded on every straight line $n = wk + v$, $n \geq 1$, $k \geq 1$, w and v are integers. This bound is expected to increase with w; by (3) surely the limit as $k \to \infty$ does. Of course $g(n, k)$ is nevertheless unbounded, which follows by the existence of the plateaus or by taking first w and then k large enough in estimate 3.

The essence of property 3 is that for large enough v, the limit along the straight line $n = wk + v$ is very close to $w+1$. Furthermore, from the crude estimate we get that $g(n, k) \leq w+1$ if $n < k(w+1)$. The increasing property of $g(n, k)$ as a function of n now establishes the fact that the graph of the function $g(n, k)$ contains an infinite series of plateaus, one for each positive integer. For each integer $w+1$, if v and μ are large enough, the values

of g at the points (n, k) fulfilling $wk + v < n$, $k > \mu$, and $(w + 1)k > n$ are in maximum norm arbitrarily close to $w + 1$. However, for large w, v may have to be chosen very large since $1 - w^{-(w+1)}$ is then very close to 1.

Globally, the function $g(n, k)$ thus has a shape resembling an infinite winding staircase with constant step height, where each step is narrower, smoother, and considerably more distant from the origin than the previous step. Closer to the origin the plateaus appear as slowly rising ridges.

REFERENCES

[1] R. A. BRUALDI AND A. J. HOFFMAN, *On the spectral radius of* (0, 1)*-matrices*, Linear Algebra Appl., 65 (1985), pp. 113–146.

[2] R. A. BRUALDI AND M. KATZ, *An extremal problem concerning matrices of* 0*'s and* 1*'s*, Linear and Multilinear Algebra, 20 (1987), pp. 325–331.

[3] R. L. GRAHAM, *Bounds on multiprocessing timing anomalies*, SIAM J. Appl. Math., 17 (1969), pp. 416–429.

[4] R. KRAHL, *Allocation problems in distributed computer systems*, Inform. Inf. Rep., 12 (1990), pp. 101–115.

[5] L. LUNDBERG, *Static process allocation using information about program behavior*, Proc. 24th Hawaii International Conference on System Sciences, Hawaii, IEEE Computer Society, 1991, pp. 1–9.

[6] L. LUNDBERG AND H. LENNERSTAD, *Comparing the performance of optimal dynamic and static process allocation*, Tech. report, Department of Computer Science and Economy, University of Karlskrona/Ronneby, Sweden, 1993.

[7] M. NADERI, *Modelling and performance evaluation of multiprocessor organisation with shared memories*, Comput. Archit. News, 16 (1988), pp. 51–74.

[8] C. C PRICE, *Task allocation in data flow multiprocessors: An annotated bibliography*, Comput. Archit. News, 19 (1991), pp. 128–134.

[9] Proc. 20th Southeastern Conference on Combinatorics, Graph Theory and Computing, Boca Raton, FL, 1989.

[10] Proc. 22nd Southeastern Conference on Combinatorics, Graph Theory and Computing, Baton Rouge, LA, 1991.

[11] J. ROST, *A distributed algorithm for dynamic task scheduling*, CONPAR 90-VAPP, N. Joint Internat. Conference on Vector and Parallel Processing, Proc., Zurich, Switzerland, 1990, Springer-Verlag, pp. 628–639.

[12] B. SHIRAZI AND M.-F. WANG, *Heuristic functions for static task allocation*, Microprocess. Microprogr. (Netherlands), 26 (1989), pp. 187–194.

[13] J. ZAHORJAN AND C. MCCANN, *Processor sheduling in shared memory multiprocessors*, Performance Evaluation, 18 (1990).

SIAM J. COMPUT.
Vol. 24, No. 4, pp. 765–796, August 1995

006

COMPUTING THE DEGREE OF DETERMINANTS VIA COMBINATORIAL RELAXATION*

KAZUO MUROTA†

Abstract. Let $A(x) = (A_{ij}(x))$ be a square matrix with A_{ij} being a polynomial in x. This paper proposes "combinatorial relaxation-" type algorithms for computing the degree of the determinant $\delta(A) = \deg_x \det A(x)$ based on its combinatorial upper bound $\widehat{\delta}(A)$, which is defined in terms of the maximum weight of a perfect matching in an associated graph. The graph is bipartite for a general square matrix A and nonbipartite for a skew-symmetric A. The algorithm transforms A to another matrix A', for which $\delta(A) = \delta(A') = \widehat{\delta}(A')$ with successive elementary operations. The algorithm is efficient, making full use of the fast algorithms for weighted matchings; it is combinatorial in almost all cases (or generically) and invokes algebraic elimination routines only when accidental numerical cancellations occur.

It is shown in passing that for a (skew-)symmetric polynomial matrix $A(x)$ there exists a unimodular matrix $U(x)$ such that $A'(x) = U(x)A(x)U(x)^{\mathrm{T}}$ satisfies $\delta(A) = \delta(A') = \widehat{\delta}(A')$.

Key words. determinant, matching, polynomial matrix, computer algebra, combinatorial optimization, polyhedral combinatorics

AMS subject classifications. 05C50, 15A15, 68Q40, 68Q25

1. Introduction.

Let $A(x) = (A_{ij}(x))$ be an $n \times n$ polynomial matrix with

$$A_{ij}(x) = \sum_{s \in \mathbf{Z}_+} A_{ijs}\, x^s, \tag{1.1}$$

where the coefficients A_{ijs} are elements of a certain field $\mathbf{F}$ (typically the real number field $\mathbf{R}$) and the summation is taken over a finite subset of nonnegative integers $\mathbf{Z}_+$. In this paper we consider computational procedures for

$$\delta(A) = \deg_x \det A(x), \tag{1.2}$$

the degree of determinant of $A(x)$. In particular, we are interested in algorithms that avoid the explicit computation of the coefficients f_s in the polynomial expansion of the determinant $f(x) = \det A(x) = \sum_s f_s\, x^s$.

This problem will arise in many different branches of the mathematical sciences. In control theory (e.g., [33], [38]), for instance, the number of exponential modes of a linear time-invariant descriptor system $E\mathrm{d}\xi/\mathrm{d}t = F\xi + Gu$ with descriptor-vector ξ and input-vector u is equal to $\delta(F - xE)$, which is sometimes called the *dynamical degree* [20], [23]. Thus, the determination of the dynamical degree of a large-scale descriptor system is a typical example of the present problem.

The present work is another attempt of the "combinatorial relaxation" approach to algebraic computations, which has been proposed recently by Murota [26] for the computation of a Newton polygon. This approach establishes a link between computer algebra [6], [9], [34] and mathematical programming (combinatorial optimization [21], [31], in particular). We make use of results in mathematical programming in two different ways. Firstly, the proposed algorithms use the results from polyhedral combinatorics in their individual steps; the correctness relies on duality theorems and the practical efficiency relies on fast combinatorial algorithms.

*Received by the editors July 16, 1991; accepted for publication (in revised form) February 26, 1994.

†Research Institute for Mathematical Sciences, Kyoto University, Kyoto 606, Japan (murota@kurims.kyoto-u.ac.jp.). This work was done while the author was at the Department of Mathematical Engineering and Information Physics, University of Tokyo, and at Forschungsinstitut für Diskrete Mathematik, Universität Bonn.

Secondly, the algorithms are designed in line with a general idea known as "relaxation" in mathematical programming, which typically appears in integer programming [35].

In general, an algorithm of "combinatorial relaxation" type consists of the following three distinct phases:

Phase 1: Consider a relaxation (or an easier problem) of a combinatorial nature to the original problem and find a solution to the relaxed problem.

Phase 2: Test for the validity of this solution to the original problem.

Phase 3 (in case of invalid solution): Modify the relaxation so that the invalid solution is eliminated.

It is important for computational efficiency that the relaxed problem can be solved efficiently and the modification of the relaxation in Phase 3 need not be invoked many times.

The proposed algorithms for $\delta(A)$ are designed based on well-known "generic" characterizations (see §§2.1 and 4.1) of $\delta(A)$ in terms of perfect matchings of a graph $G(A)$ associated with the given matrix A. Here the word "generic" refers to an algebraic assumption that the nonzero coefficients A_{ijs} in (1.1) are subject to no algebraic relations, but its practical interpretation would be "so long as no accidental numerical cancellation occurs."

For a general nonsymmetric matrix A, the associated graph $G(A)$ is a bipartite graph $G_0(A)$, each edge of which corresponds to a nonzero entry $A_{ij}(x)$ and is associated with $\deg A_{ij}(x)$ as a weight. The maximum weight $\widehat{\delta}_0(A)$ of a perfect matching in $G_0(A)$ is equal to the highest degree of a nonzero term in the determinant expansion, which is an upper bound on $\delta(A)$ (i.e., $\widehat{\delta}_0(A) \geq \delta(A)$) and is generically equal to $\delta(A)$.

For a skew-symmetric matrix A, $\widehat{\delta}_0(A)$ is not qualified as a generic characterization of $\delta(A)$ because of the nonnumerical cancellation of terms resulting from skew-symmetry. In this case, $\delta(A)$ is generically equal to twice the highest degree of a nonzero term in the Pfaffian [22] and the relevant combinatorial object is known to be matching in a nonbipartite graph $G_1(A)$. In fact, this connection of nonbipartite matchings and skew-symmetric matrices was used in the original proof of Tutte's theorem [37]. Let $\widehat{\delta}_1(A)$ denote twice the maximum weight of a perfect matching in $G_1(A)$. Then $\widehat{\delta}_1(A) \geq \delta(A)$ and the equality holds generically.

The proposed algorithm first finds $\widehat{\delta}(A)$ (either $\widehat{\delta}_0(A)$ or $\widehat{\delta}_1(A)$) instead of $\delta(A)$ by solving a weighted-matching problem using an efficient combinatorial algorithm (Phase 1) and then tests for nonsingularity of a constant matrix to see whether $\widehat{\delta}(A)$ equals $\delta(A)$ (Phase 2). The algorithm invokes an exception-handling algebraic elimination routine to modify A only when it detects a discrepancy between $\widehat{\delta}(A)$ and $\delta(A)$ resulting from numerical cancellation (Phase 3). Since numerical cancellation occurs only rarely (or nongenerically), the proposed algorithm is in almost all cases combinatorial and hence suitable for large scale problems.

In Phase 3 the matrix A (with $\delta(A) \leq \widehat{\delta}(A) - 1$) is modified to another matrix A' such that $\delta(A') = \delta(A)$ and $\widehat{\delta}(A') \leq \widehat{\delta}(A) - 1$. The modification algorithm makes essential use of dual variables based on the duality theorem for the polyhedral description of matchings of the following kind (see §§2.2 and 4.2):

$$\begin{aligned}\widehat{\delta}(A) &= \max\{\text{weight of a matching}\}\\ &= \max\{\text{primal IP}(A)\} \leq \max\{\text{primal LP}(A)\}\\ &= \min\{\text{dual LP}(A)\} \leq \min\{\text{dual IP}(A)\},\end{aligned}$$

where LP(A) and IP(A) stand for the linear program and the integer program for the matching problem in $G(A)$, respectively. The integrality theorems (Edmonds [12], Cunningham and Marsh [8]) state that all these "obvious" inequalities are in fact equalities. Combining this with $\delta(A) \leq \widehat{\delta}(A)$, in which equality holds generically, we obtain

$$\delta(A) \leq \widehat{\delta}(A) = \min\{\text{dual IP}(A)\}, \tag{1.3}$$

which is satisfied with equality in the generic case, as an extension of the duality principle. The modification algorithm of Phase 3, which relies on the duality (1.3) above, amounts to establishing a novel identity

$$\delta(A) = \min\{\widehat{\delta}(A') \mid A' \in \mathcal{M}(A)\}, \tag{1.4}$$

where $\mathcal{M}(A) = \{U(x)A(x) \mid U(x) \text{ is unimodular}\}$ for a nonsymmetric A and $\mathcal{M}(A) = \{U(x)A(x)U(x)^{\mathrm{T}} \mid U(x) \text{ is unimodular}\}$ for a skew-symmetric A (see Propositions 3.3 and 4.14). Note that $\delta(A) = \delta(A')$ for all $A' \in \mathcal{M}(A)$. Such identity will enforce the link between linear algebra (determinant) and graph/matroid theory (matching) observed in various contexts [5], [13], [18], [23], [32].

This paper is organized as follows. In §2 general nonsymmetric matrices are treated by means of bipartite matchings. This is extended to symmetric matrices in §3. The main body of this paper is in §4, where skew-symmetric matrices are treated by means of nonbipartite matchings. The complexity issues are discussed in §5.

2. General nonsymmetric matrices. In this section we consider a general $n \times n$ matrix $A(x) = (A_{ij}(x))$ with $A_{ij}(x) \in \mathbf{F}[x, 1/x]$, i.e.,

$$A_{ij}(x) = \sum_{s \in \mathbf{Z}} A_{ijs}\, x^s, \tag{2.1}$$

where $A_{ijs} \in \mathbf{F}$ (a field) and the summation is taken over a finite subset of integers. We allow negative powers in x, though we are mainly interested in polynomial matrices (with $A_{ij}(x) \in \mathbf{F}[x]$). For $S = \mathbf{F}[x]$ or $S = \mathbf{F}[x, 1/x]$ the set of $n \times n$ matrices over S will be designated by $\mathcal{M}_n(S)$ or simply by $\mathcal{M}(S)$. As in (1.2), we denote by $\delta(A)$ the highest degree of a nonzero term in $\det A(x) \in \mathbf{F}[x, 1/x]$; note, however, that $\delta(A)$ can be negative and $\delta(A) = -\infty$ if $\det A(x) = 0$.

Most of the results of this section can be found in the unpublished report of Murota [25] and are implicit in [26]. They are, however, included here in an explicit form for a clear exposition of the central ideas of the present paper without involving too many technicalities, which will be introduced later in §§3 and 4.

2.1. Generic characterization of degree of determinant. The structure of A is conveniently represented by a bipartite graph $G = G(A) = G_0(A)$ defined as follows: The vertex set $V = V(G)$ is the disjoint union of the row set R and the column set C of A, and the edge set $E = E(G)$ is identified with the nonzero entries of A, i.e.,

$$E(G) = \{(i, j) \mid i \in R, j \in C, A_{ij}(x) \neq 0\}.$$

To edge $e = (i, j) \in E$ is attached a cost (or weight)

$$c_e = c_{ij} = \max\{s \mid A_{ijs} \neq 0\} = \deg_x A_{ij}(x). \tag{2.2}$$

The set of end vertices of an edge $e = (i, j) \in E$ is denoted as $\partial e = \{i, j\}$, and this notation is extended for $M \subseteq E$ as $\partial M = \bigcup\{\partial e \mid e \in M\}$. A subset M of E is called a matching if $|\partial M| = 2|M|$ and a perfect matching if $|\partial M| = 2|M| = |V|$. We denote by $\mu(G)$ the maximum size of a matching in G. The cost (or weight) of $M \subseteq E$ is defined by

$$c(M) = \sum_{(i,j) \in M} c_{ij}.$$

M is an optimal matching if M is a perfect matching of maximum weight.

We now introduce $\widehat{\delta}_0(A)$, which plays the central role as a combinatorial counterpart of $\delta(A)$. We define

$$\widehat{\delta}_0(A) = \max\{c(M) \mid M \text{ is a perfect matching in } G_0(A)\}, \tag{2.3}$$

where $\widehat{\delta}_0(A) = -\infty$ if no perfect matching exists. It is easy to see that $\widehat{\delta}_0(A)$ is equal to the highest degree of a nonzero term in the defining expansion of the determinant

$$\det A(x) = \sum_{\sigma} \operatorname{sgn}\sigma \prod_{i=1}^{n} A_{i\sigma(i)}(x). \tag{2.4}$$

On recognizing a perfect matching as an alias of a permutation, we obtain the following proposition.

PROPOSITION 2.1. *Let $A(x)$ be a square matrix.*

(1) $\delta(A) \leq \widehat{\delta}_0(A)$.

(2) *The equality holds generically, i.e., if the set of nonzero leading coefficients $\{A_{ijs} \mid A_{ij}(x) \neq 0, s = \deg A_{ij}(x)\}$ is algebraically independent (over a subfield of* **F**). □

This observation means that we may expect $\widehat{\delta}_0(A)$ to be equal to $\delta(A)$ if accidental numerical cancellation does not occur. We say that $A(x)$ is *upper tight* if $\delta(A) = \widehat{\delta}_0(A)$.

Our algorithm takes advantage of the available fast combinatorial algorithms (see [1], [19], [21], [31]) to compute $\widehat{\delta}_0(A)$, which serves as a first approximation to $\delta(A)$. The outline of the proposed algorithm of "combinatorial relaxation" type follows. The detailed procedures of Phase 2 and Phase 3 will be described later.

ALGORITHM FOR COMPUTING $\delta(A)$ (OUTLINE).

Phase 1: Compute $\widehat{\delta}_0(A)$ by solving the weighted-matching problem in $G(A)$ using an efficient combinatorial algorithm.

Phase 2: Test whether or not $\delta(A) = \widehat{\delta}_0(A)$ (without computing $\delta(A)$).
If so, output $\widehat{\delta}_0(A)$ as $\delta(A)$ and stop.

Phase 3: Modify A to another matrix A' such that $\delta(A') = \delta(A)$ and $\widehat{\delta}_0(A') \leq \widehat{\delta}_0(A) - 1$.
Put $A := A'$ and go to Phase 1.

Example 2.1. Consider the following matrix:

$$A(x) = \begin{pmatrix} \alpha & 0 & x^2 \\ x & 1 & x^3 \\ 1 & 1 & 1 \end{pmatrix}$$

with α being a nonzero parameter (free from x).

The associated bipartite graph $G = G_0(A)$ is depicted in Fig. 1. It has six vertices and eight edges, and admits four perfect matchings with weights 3, 3, 2, and 0. By direct expansion we obtain

$$\det A(x) = -\alpha x^3 + x^3 - x^2 + \alpha = (1-\alpha)x^3 - x^2 + \alpha.$$

Accordingly, we have

$$\widehat{\delta}_0(A) = 3, \quad \delta(A) = \begin{cases} 3 & \text{if } \alpha \neq 1, \\ 2 & \text{if } \alpha = 1. \end{cases}$$ □

2.2. Test for upper tightness. This section describes a procedure for Phase 2 which tests for the upper tightness (i.e., $\delta(A) = \widehat{\delta}_0(A)$) of $A(x)$ without expanding $\det A(x)$. The procedure makes use of the standard duality results for bipartite matchings, which follow from the integrality of the associated linear program.

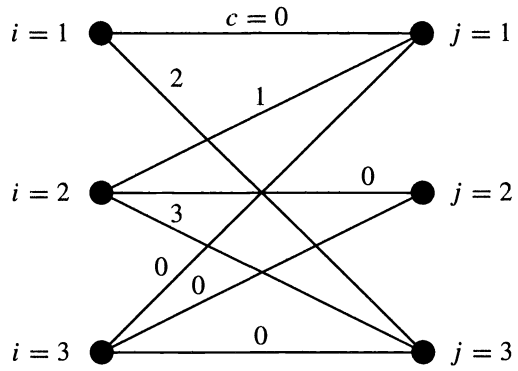


FIG. 1. *Bipartite graph* $G_0(A)$ (*Example* 2.1).

Consider a linear program

$$\text{PLP: maximize} \quad \sum_{e \in E} c_e \xi_e,$$

$$\text{subject to} \quad \sum_{\partial e \ni i} \xi_e = 1 \quad (i \in V), \tag{2.5}$$

$$\xi_e \geq 0 \quad (e \in E),$$

and its dual

$$\text{DLP: minimize} \quad \sum_{i \in V} p_i \quad (\equiv \pi(p)),$$

$$\text{subject to} \quad p_i + p_j \geq c_{ij} \quad ((i, j) \in E). \tag{2.6}$$

Note that $\xi = (\xi_e \mid e \in E)$ is the primal variable and $p = (p_i \mid i \in V) = (p_{Ri} \mid i \in R) \oplus (p_{Cj} \mid j \in C)$ is the dual variable.

As is well known, these linear programs enjoy the integrality property.

PROPOSITION 2.2. (1) *PLP has an integral optimal solution* (*with* $\xi_e \in \{0, 1\}$ $(e \in E)$).

(2) *If* c_e *is an integer for* $e \in E$, *DLP has an integral optimal solution* (*with* $p_i \in \mathbf{Z}$ $(i \in V)$). □

This implies that

$$\widehat{\delta}_0(A) = \min\{\pi(p) \mid p \text{ is feasible to DLP}\}. \tag{2.7}$$

The optimality of a perfect matching is expressed by the complementary slackness condition as follows: For $e = (i, j) \in E$,

$$\tilde{c}_e = \tilde{c}_{ij} = c_{ij} - p_i - p_j \tag{2.8}$$

is called the reduced cost with respect to p. Then p is (dual) feasible if and only if $\tilde{c}_e \leq 0$ $(e \in E)$. An edge e is said to be tight (with respect to p) if $\tilde{c}_e = 0$. We set

$$E^* = E^*(p) = \{e \in E \mid \tilde{c}_e = 0\},$$

which is the set of tight edges. We also set $G^* = G^*(p) = (V, E^*(p))$.

PROPOSITION 2.3. *Let* M *be a perfect matching in* $G(A)$ *and* p *be a dual feasible solution. Then both* M *and* p *are optimal if and only if* $M \subseteq E^*(p)$. □

The following corollary is important for our algorithm. Note that $E^*(p)$ depends on the choice of p.

PROPOSITION 2.4. *Let p be a dual optimal solution and M be a subset of E. Then M is an optimal matching in G if and only if M is a perfect matching in $G^*(p)$.* □

In accordance with E^* we extract the "tight" part from $A(x)$. Namely, for a dual feasible p we define

$$\mathcal{T}(A; p) = A^* = (A^*_{ij}), \quad A^*_{ij} = \begin{cases} A_{ij}c_{ij} & \text{if } (i, j) \in E^*(p), \\ 0 & \text{otherwise.} \end{cases} \tag{2.9}$$

That is,

$$A_{ij}(x) = x^{p_i + p_j}(A^*_{ij} + o(1)), \tag{2.10}$$

where $o(1)$ denotes an expression consisting of negative powers of x. Note that A^* is a constant matrix and that it depends on p.

The linear algebraic significance of the dual variables is made clearer by the "leveling" or "scaling" operation $\mathcal{L}(A; p)$ defined by

$$\mathcal{L}(A; p) = \operatorname{diag}(x; -p_R) \cdot A(x) \cdot \operatorname{diag}(x; -p_C), \tag{2.11}$$

where, for a vector $r = (r_i \mid i = 1, \ldots, n)$, in general,

$$\operatorname{diag}(x; r) = \operatorname{diag}(x^{r_1}, x^{r_2}, \ldots, x^{r_n}).$$

Note that $\mathcal{L}(A; p) \in \mathcal{M}(\mathbf{F}[x, 1/x])$ if p is integral.

PROPOSITION 2.5. *Let $\tilde{A}(x) = \mathcal{L}(A; p)$ and $\pi(p) = \sum_{i \in V} p_i$.*

(1) $\delta(\tilde{A}) = \delta(A) - \pi(p)$, $\hat{\delta}_0(\tilde{A}) = \hat{\delta}_0(A) - \pi(p)$.

(2) *If p is dual feasible, then $\tilde{A}(x) = A^* + o(1)$.*

Proof. (1) The first relation is immediate from (2.11). The second follows from $\sum_{(i,j) \in M}(c_{ij} - p_i - p_j) = c(M) - \sum_{i \in V} p_i$, which is true for any perfect matching M.

(2) This is a restatement of (2.10). □

The following proposition shows that the test for upper tightness of $A(x)$ is reduced to the test for nonsingularity of a constant matrix A^*. It is emphasized that an integer-valued dual optimal solution can be computed efficiently.

PROPOSITION 2.6. *Let p be a dual optimal solution and $A^* = \mathcal{T}(A; p)$.*

(1) $\det A(x) = x^{\hat{\delta}_0(A)}(\det A^* + o(1))$. *In particular, $\det A^*$ is independent of the choice of p, although the matrix A^* itself depends on p.*

(2) *$A(x)$ is upper tight (i.e., $\delta(A) = \hat{\delta}_0(A)$) if and only if A^* is nonsingular.*

Proof. It follows from Proposition 2.5 (1) that $A(x)$ is upper tight if and only if $\tilde{A}(x) = \mathcal{L}(A; p)$ is upper tight. Note that $A^* = \mathcal{T}(A; p) = \mathcal{T}(\tilde{A}; 0)$. Then Proposition 2.5 (2) implies

$$\det \tilde{A}(x) = \det A^* + o(1).$$

This completes the proof since $\hat{\delta}_0(\tilde{A}) = 0$ by the optimality of p. □

Remark 2.1. In general, $\mu(G_0(A^*))$(= maximum size of a matching in $G_0(A^*)$) is called the term-rank of A^*, and term-rank $A^* \geq$ rank A^* with equality in the generic case. By construction, term-rank $A^* = n$ for p dual optimal. Hence the above proposition may be rephrased as follows: $A(x)$ is upper tight if and only if term-rank A^* = rank A^*. □

Example 2.2 (continued from Example 2.1). As the optimal dual variables we may take

$$p_{R1} = 0, \quad p_{R2} = 1, \quad p_{R3} = 0; \quad p_{C1} = 0, \quad p_{C2} = 0, \quad p_{C3} = 2.$$

Those variables and the reduced costs are illustrated in Fig. 2.

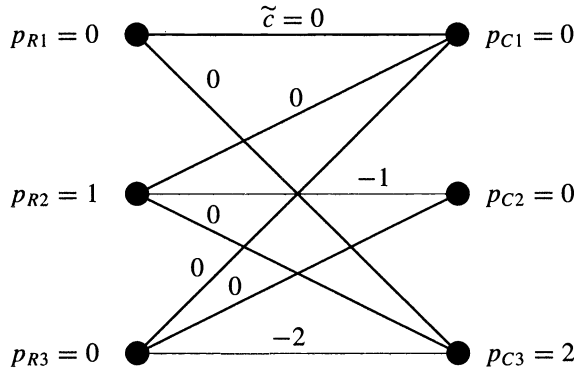


FIG. 2. *Reduced costs and dual variables for* $G_0(A)$ *(Example* 2.2).

We have $\widehat{\delta}_0(A) = \pi(p) = 3$. According to (2.9) we have

$$A^* = \begin{pmatrix} \alpha & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}. \tag{2.12}$$

Proposition 2.6 shows that $A(x)$ is upper tight for $\alpha \neq 1$ since $\det A^* = 1 - \alpha$. □

2.3. Transformation to an upper-tight matrix. When the matrix $A(x)$ is not upper tight, the combinatorial characteristic $\widehat{\delta}_0(A)$ gives only an upper bound on $\delta(A)$. In this section we will show how to transform efficiently $A(x)$ to an upper-tight matrix through repeated unimodular row transformations. Note that the Hermite normal form (see, e.g., [30], [35]) guarantees the existence of such an upper-tight matrix, though this fact is not used below.

Given $A(x) \in \mathcal{M}(\mathbf{F}[x, 1/x])$ with $\delta(A) < \widehat{\delta}_0(A)$, we must modify $A(x)$ to another matrix $A'(x) = (A'_{ij}(x))$ such that

(P1) $A'(x) = U(x)A(x)$ with $U(x) \in \mathcal{M}(\mathbf{F}[x, 1/x])$, $\det U(x) = 1$;

(P2) $\widehat{\delta}_0(A') \leq \widehat{\delta}_0(A) - 1$.

In particular, (P1) implies $\delta(A') = \delta(A)$. It should be obvious that we can get an upper-tight matrix by repeatedly applying this transformation.

When $A(x)$ is a polynomial matrix (belonging to $\mathcal{M}(\mathbf{F}[x])$), it is natural to require that $U(x)$ be a polynomial matrix in (P3a) below. On the other hand, (P3b) will turn out (see §5) to be convenient for bounding the worst-case complexity. Thus we impose either of the following additional conditions:

(P3a) $U(x) \in \mathcal{M}(\mathbf{F}[x])$,

(P3b) $U(x) \in \mathcal{M}(\mathbf{F}[1/x])$.

Recall that a constant matrix A^* is derived from $A(x)$ with reference to an optimal dual variable $p = \{p_{Ri}, p_{Cj}\}$, which is assumed to be integer valued. We have term-rank $A^* = n$, while rank $A^* < n$, since $A(x)$ is not upper tight (cf. Proposition 2.6 and Remark 2.1). Then there exists (cf. (2.19) below) a nonsingular constant matrix $U = (U_{ik})$ with $\det U = 1$ such that

$$\text{term-rank}\,(UA^*) < n. \tag{2.13}$$

Using the dual variables $\{p_{Ri}, p_{Cj}\}$ for $G_0(A)$, we define the transformation from A to A' by

$$A'(x) = U(x)A(x), \tag{2.14}$$

where $U(x) = (U_{ik}(x))$ is given by

$$U_{ik}(x) = U_{ik}\, x^{\sigma(i,k)}, \quad \sigma(i,k) = p_{Ri} - p_{Rk}. \tag{2.15}$$

Note that

$$U(x) = \mathrm{diag}\,(x; p_R) \cdot U \cdot \mathrm{diag}\,(x; -p_R),$$

and hence (2.14) is equivalent to

$$\widetilde{A}'(x) = U\widetilde{A}(x) \tag{2.16}$$

in terms of $\widetilde{A}(x) = \mathcal{L}(A; p)$ and $\widetilde{A}'(x) = \mathcal{L}(A'; p)$. (P1) is satisfied since p is integer valued.

We claim that property (P2) is satisfied.

PROPOSITION 2.7. *Assume that p is integral dual optimal.* (P2) *holds if U satisfies* (2.13).

Proof. Since

$$\widetilde{A}'(x) = U\widetilde{A}(x) = UA^* + O(1/x)$$

and term-rank $(UA^*) < n$ by (2.13), we see $\widehat{\delta}_0(\widetilde{A}') \leq -1$. By Proposition 2.5 this is equivalent to

$$\widehat{\delta}_0(A') \leq \pi(p) - 1 = \widehat{\delta}_0(A) - 1,$$

where $\pi(p)$ is defined by (2.7). □

As for property (P3), we easily see the following.

PROPOSITION 2.8. *Assume that p is integral dual optimal.*

(1) (P3a) *holds if* $[U_{ik} \neq 0 \implies p_{Ri} \geq p_{Rk}]$.

(2) (P3b) *holds if* $[U_{ik} \neq 0 \implies p_{Ri} \leq p_{Rk}]$. □

The above statement says, in effect, that U should be in a triangular form if its rows and columns are rearranged according to the orderings determined by the dual variables associated with the rows of $A(x)$.

A concrete choice of U that meets conditions (P1), (P2), and (P3) is now suggested. Since rank $A^* < n$, there exists a nonzero vector $u = (u_i \mid i \in R)$ (indexed by the rows) such that

$$u^{\mathrm{T}}A^* = 0. \tag{2.17}$$

We choose u with minimal support, i.e., such that $\mathrm{supp}\, u = \{i \in R \mid u_i \neq 0\}$ is minimal with respect to set inclusion. (Such u can be computed by the Gaussian elimination on A^* with column pivoting.) Let $h \in \mathrm{supp}\, u$ be such that either

$$\text{(a) } p_{Rh} = \max\{p_{Ri} \mid i \in \mathrm{supp}\, u\} \text{ or (b) } p_{Rh} = \min\{p_{Ri} \mid i \in \mathrm{supp}\, u\}, \tag{2.18}$$

and define U by

$$U_{ik} = \begin{cases} u_k/u_h & \text{if } i = h, \\ \delta_{ik} & \text{otherwise,} \end{cases} \tag{2.19}$$

where δ_{ik} denotes the Kronecker delta ($\delta_{ik} = 1$ for $i = k$ and $= 0$ otherwise). The additional condition (P3a) or (P3b) is satisfied accordingly as h is chosen by the criteria (a) or (b) in (2.18).

Example 2.3 (continued from Example 2.2). Consider the case of $\alpha = 1$, where A^* of (2.12) is singular. We may take $u = (-1, 1, 0)^{\mathrm{T}}$ with $\mathrm{supp}\, u = \{1, 2\}$. According to the criterion (a) we have $h = 2$ and

$$U = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Then the matrix A is modified to

$$A'(x) = \mathrm{diag}\,(1, x, 1) \cdot U \cdot \mathrm{diag}\,(1, x^{-1}, 1) \cdot A(x) = \begin{pmatrix} 1 & 0 & x^2 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix},$$

which is upper tight with $\widehat{\delta}_0(A') = \delta(A') = 2$. □

2.4. Description of algorithm. Combining the procedures given above, we obtain the following algorithm for computing $\delta(A)$ for $A(x) \in \mathcal{M}(\mathbf{F}[x, 1/x])$.

ALGORITHM FOR COMPUTING $\delta(A)$ (A: NONSYMMETRIC MATRIX).

Step 0

$c_{\min} := \min_{i,j} \min\{s \mid A_{ijs} \neq 0\}$.

Step 1

(1): Find a maximum weight perfect matching M and integer-valued optimal dual variables p_{Ri} ($i \in R$), p_{Cj} ($j \in C$) for $G_0(A)$;
$\widehat{\delta}_0(A) := c(M)$ ($\widehat{\delta}_0(A) = -\infty$ if no perfect matching exists).

(2): If $\widehat{\delta}_0(A) < n \cdot c_{\min}$, then stop with $\delta(A) := -\infty$.

Step 2

(1): $A^*_{ij} :=$ coefficient of $x^{p_{Ri}+p_{Cj}}$ in $A_{ij}(x)$ ($i \in R$, $j \in C$). [cf.(2.9)]

(2): If $\det A^* \neq 0$, then stop with $\delta(A) := \widehat{\delta}_0(A)$.

Step 3 [$\det A^* = 0$]

(1): Find u with minimal support such that $u^{\mathrm{T}}A^* = 0$. [cf.(2.17)]

(2): Let $h \in \mathrm{supp}\, u$ be such that either

(a): $p_{Rh} = \max\{p_{Ri} \mid i \in \mathrm{supp}\, u\}$ or (b): $p_{Rh} = \min\{p_{Ri} \mid i \in \mathrm{supp}\, u\}$.

(3): $s := u_h$; $u_i := u_i/s$ ($i = 1, \ldots, n$).

(4):

$$A_{ij}(x) := \begin{cases} \sum_k x^{\sigma(h,k)} u_k A_{kj}(x) & \text{if } i = h,\ j \in C, \\ A_{ij}(x) & \text{otherwise,} \end{cases}$$

where $\sigma(h, k) = p_{Rh} - p_{Rk}$. [cf.(2.14), (2.19)]

(5): Go to Step 1.

It follows from (P2) that the number of iterations in the above algorithm is bounded by $\widehat{\delta}_0(A^{(0)}) - \delta(A^{(0)})$ if $\delta(A^{(0)}) \neq -\infty$, where $A^{(0)}$ denotes the input matrix. Note that, in general, $\widehat{\delta}_0(A) \leq n \cdot c_{\max}(A)$, where

$$c_{\max}(A) = \max\{\deg_x A_{ij}(x) \mid i \in R,\ j \in C\}.$$

The stopping criterion in Step 1 (2) is to cope with the case of $\delta(A^{(0)}) = -\infty$. In Step 2, we need row elimination operations on A^*. Though it requires $O(n^3)$ arithmetic operations in $\mathbf{F}$ in the worst case, it can be done much faster since A^* is usually very sparse in practical applications. Other worst-case complexity issues will be discussed in §5.

Let us consider the probabilistic behavior of the algorithm. As already noted in Proposition 2.1, $\widehat{\delta}_0(A)$ differs from $\delta(A)$ only because of accidental numerical cancellation. Let us fix the structure (i.e., the index set $\{(i, j, s) \mid A_{ijs} \neq 0\}$ of nonzero coefficients in (2.1)) of the input matrix $A = A^{(0)}$ and assume that the numerical values of coefficients $A_{ijs} \in \mathbf{R} = \mathbf{F}$ can be modeled as real-valued random variables with continuous distributions. Then we have $\widehat{\delta}_0(A) = \delta(A)$ with probability one, which means that Step 3 is performed only with null

probability. Since the worst-case time complexity for the assignment problem is bounded by $O(n^3)$, we obtain the following statement, indicating the practical efficiency of the proposed algorithm. The average time complexity of the proposed algorithm (in the above sense) is bounded by a polynomial in n (e.g., n^3).

3. Symmetric matrices. In this section we consider a symmetric matrix $A(x) = A(x)^{\mathrm{T}} \in \mathcal{M}_n(\mathbf{F}[x, 1/x])$ based on the result of Murota [28]. Symmetric matrices can be treated in a way quite similar to the previous section by means of bipartite matchings, as opposed to skew-symmetric matrices for which nonbipartite matchings must be employed (cf. §4). A new feature with a symmetric matrix is that in Phase 3, a congruence transformation $A'(x) = U(x)A(x)U(x)^{\mathrm{T}}$ should be used to preserve the symmetry.

3.1. Generic characterization of degree of determinant. For a symmetric matrix $A(x)$ we again employ the bipartite graph $G = G(A) = G_0(A)$ and $\widehat{\delta}_0(A)$ of (2.3) as defined in §2. Proposition 2.1 remains valid with an obvious modification in the second statement and $\widehat{\delta}_0(A)$ serves as a generic characterization of $\delta(A)$ in spite of the algebraic dependency of the coefficients resulting from symmetry.

PROPOSITION 3.1. *Let $A(x)$ be a symmetric matrix.*

(1) $\delta(A) \leq \widehat{\delta}_0(A)$.

(2) *The equality holds generically, i.e., if the set of nonzero leading coefficients $\{A_{ijs} \mid A_{ij}(x) \neq 0, s = \deg A_{ij}(x), i \leq j\}$ is algebraically independent (over a subfield of $\mathbf{F}$), provided $\mathbf{F}$ is not of characteristic two.*

Proof. (1) This is a special case of Proposition 2.1 (1).

(2) In the determinant expansion (2.4) there appear similar terms, which, however, never cancel each other, having the same sign. □

We can design an algorithm of "combinatorial relaxation" type, as outlined in §2.1, on the basis of $\widehat{\delta}_0(A)$. Whereas Steps 1 and 2 of the algorithm of §2.4 remain valid, Step 3 needs to be adapted to symmetry.

3.2. Transformation to an upper-tight matrix. We will show how to transform $A(x)$ to an upper-tight matrix through repeated unimodular congruence transformations. The existence of such an upper-tight matrix itself does not seem obvious (see Proposition 3.3 below).

Given a symmetric $A(x) \in \mathcal{M}(\mathbf{F}[x, 1/x])$ with $\delta(A) < \widehat{\delta}_0(A)$, we must modify $A(x)$ to another matrix $A'(x) = (A'_{ij}(x))$ such that

(P1-S) $A'(x) = U(x)A(x)U(x)^{\mathrm{T}}$ with $U(x) \in \mathcal{M}(\mathbf{F}[x, 1/x])$, $\det U(x) = 1$,

and (P2) of §2.3 are satisfied. Furthermore, we sometimes impose either (P3a) or (P3b) of §2.3.

We assume that $p = \{p_{Ri}, p_{Cj}\}$ is integer-valued dual optimal. Define a new dual variable $p' = \{p'_{Ri}, p'_{Cj}\}$ by $p'_{Ri} = p'_{Ci} = \bar{p}_i$ $(i = 1, \ldots, n)$, where

$$\bar{p}_i = (p_{Ri} + p_{Ci})/2 \qquad (i = 1, \ldots, n).$$

Noting that p' is also dual optimal we consider $A^* = \mathcal{T}(A; p')$ with respect to which we choose u according to (2.17) and define U by (2.19) and $U(x)$ by (2.15), where h is determined by (2.18) with p replaced by p'. Properties (P2) and (P3) are satisfied as in §2.3, whereas (P1-S) needs a separate consideration since p' may not be integer valued.

PROPOSITION 3.2. (P1-S) *is satisfied if p is integer-valued.*

Proof. For $(i, j) \in E^*(p')$ we have $\bar{p}_i - \bar{p}_j \in \mathbf{Z}$ since $\bar{p}_i \in \frac{1}{2}\mathbf{Z}$, $\bar{p}_j \in \frac{1}{2}\mathbf{Z}$, and $\bar{p}_i + \bar{p}_j = c_{ij} \in \mathbf{Z}$. The minimality of $\operatorname{supp} u$ implies that $\bar{p}_i - \bar{p}_j \in \mathbf{Z}$ for $i, j \in \operatorname{supp} u$ (cf. [28, proof of Lem. 7]). Hence $U(x)$ of (2.15) belongs to $\mathcal{M}(\mathbf{F}[x, 1/x])$. □

3.3. Description of algorithm. The following algorithm is obtained for computing $\delta(A)$ for a symmetric $A(x) \in \mathcal{M}(\mathbf{F}[x, 1/x])$.

ALGORITHM FOR COMPUTING $\delta(A)$ (A: SYMMETRIC MATRIX).

Step 0

$c_{\min} := \min_{i,j} \min\{s \mid A_{ijs} \neq 0\}$.

Step 1

(1): Find a maximum weight perfect matching M and integer-valued optimal dual variables p_{Ri} ($i \in R$), p_{Cj} ($j \in C$) for $G_0(A)$;
$\widehat{\delta}_0(A) := c(M)$ ($\widehat{\delta}_0(A) = -\infty$ if no perfect matching exists).

(2): If $\widehat{\delta}_0(A) < n \cdot c_{\min}$, then stop with $\delta(A) := -\infty$.

Step 2

(1): $\bar{p}_i := (p_{Ri} + p_{Ci})/2 \quad (i = 1, \ldots, n)$.

(2): $A^*_{ij} :=$ coefficient of $x^{\bar{p}_i + \bar{p}_j}$ in $A_{ij}(x) \quad (i \in R, j \in C)$. [cf. (2.10)]

(3): If $\det A^* \neq 0$, then stop with $\delta(A) := \widehat{\delta}_0(A)$.

Step 3 [$\det A^* = 0$]

(1): Find u with minimal support such that $u^{\mathrm{T}} A^* = 0$. [cf. (2.17)]

(2): Let $h \in \operatorname{supp} u$ be such that either

$$\text{(a)}: \bar{p}_h = \max\{\bar{p}_i \mid i \in \operatorname{supp} u\} \quad \text{or} \quad \text{(b)}: \bar{p}_h = \min\{\bar{p}_i \mid i \in \operatorname{supp} u\}.$$

(3): $s := u_h$; $u_i := u_i/s \quad (i = 1, \ldots, n)$.

(4):

$$A_{ij}(x) := \begin{cases} \sum_k \sum_l x^{\sigma(h,k)+\sigma(h,l)} u_k u_l A_{kl}(x) & \text{if } i = j = h, \\ \sum_k x^{\sigma(h,k)} u_k A_{kj}(x) & \text{if } (i = h, j \neq h) \\ & \text{or } (j = h, i \neq h), \\ A_{ij}(x) & \text{otherwise,} \end{cases}$$

where $\sigma(h, k) = \bar{p}_h - \bar{p}_k$. [cf. (P1-S)]

(5): Go to Step 1.

Since this algorithm (with the criterion (a) in Step 3 (2)) terminates after a finite number of steps, it gives a proof for the following existence theorem.

PROPOSITION 3.3 [28]. *For a nonsingular symmetric polynomial matrix $A(x) \in \mathcal{M}(\mathbf{F}[x])$ there exists a unimodular matrix $U(x) \in \mathcal{M}(\mathbf{F}[x])$ such that $A'(x) = U(x)A(x)U(x)^{\mathrm{T}}$ is upper tight, i.e.,* $(\delta(A) =)\ \delta(A') = \widehat{\delta}_0(A')$. □

4. Skew-symmetric matrices. This section is the main part of the present paper and considers an $n \times n$ skew-symmetric matrix $A(x) = -A(x)^{\mathrm{T}} \in \mathcal{M}_n(\mathbf{F}[x, 1/x])$ of the form (2.1), where it is assumed that n is even and the characteristic of $\mathbf{F}$ is distinct from two. Since nonnumerical (or combinatorial) cancellations of terms happen in the determinant expansion because of skew-symmetry, $\widehat{\delta}_0(A)$ is no longer qualified as a generic substitute for $\delta(A)$. To cope with this nonnumerical, as well as accidental numerical, cancellation we employ the standard apparatus of nonbipartite matchings. The algorithm shares the same technique with [27], while it contains some additional features (described in §§4.2 and 4.4) for algorithmic efficiency. Our algorithm makes substantial use of dual variables, and the integrality and the total dual (half-) integrality of the perfect matching polytope of Edmonds [11], [12] and Cunningham and Marsh [8] play the crucial role.

4.1. Generic characterization of degree of determinant. As already observed by Tutte in his pioneering work [37] and nicely expounded by Lovász and Plummer [22], the combinatorial structure of a skew-symmetric matrix A is represented by a nonbipartite graph $G = G(A) = G_1(A)$ defined as follows: The vertex set $V = V(G)$ is identified with the row set R of A, which in turn has a natural one-to-one correspondence with the column set C;

$|V| = |R| = |C| = n$. The edge set $E = E(G)$ is identified with the nonzero entries of A, i.e.,

$$E(G) = \{(i, j) \mid i \in V, j \in V, A_{ij}(x) \neq 0\},$$

where (i, j) and (j, i) are not distinguished, and hence G has no parallel edges. An edge $e = (i, j) \in E$ is associated with the cost $c_{ij} = \deg_x A_{ij}(x)$ that represents the degree of the corresponding entry.

For a skew-symmetric A of even order the Pfaffian of A is defined by

$$\mathrm{pf}A = \sum_P a_P,$$

where the summation is taken over all partitions $P = \{\{i_1, j_1\}, \ldots, \{i_\nu, j_\nu\}\}$ $(\nu = n/2)$ of $V = \{1, \ldots, n\}$ into unordered pairs and

$$a_P = \mathrm{sgn}\begin{pmatrix} 1 & 2 & \ldots & 2\nu-1 & 2\nu \\ i_1 & j_1 & \ldots & i_\nu & j_\nu \end{pmatrix} \prod_{k=1}^{\nu} A_{i_k j_k}.$$

The following relation is well known.

PROPOSITION 4.1. *For a skew-symmetric A of even order,* $\det A = (\mathrm{pf}A)^2$. □

In place of $\widehat{\delta}_0(A)$ of (2.3) we consider

$$\widehat{\delta}_1(A) = 2 \cdot \max\{c(M) \mid M \text{ is a perfect matching in } G_1(A)\}. \tag{4.1}$$

By convention we put $\widehat{\delta}_1(A) = -\infty$ if no perfect matching exists in $G_1(A)$. Note the distinction between $\widehat{\delta}_0(A)$ and $\widehat{\delta}_1(A)$; the former is equal to the highest degree of a term in the defining expansion of $\det A$, while the latter is twice the highest degree of a term in $\mathrm{pf}A$.

The following proposition parallels Propositions 2.1 and 3.1.

PROPOSITION 4.2. *Let $A(x)$ be a skew-symmetric matrix.*

(1) $\delta(A) \leq \widehat{\delta}_1(A)$.

(2) *The equality holds generically, i.e., if the set of nonzero leading coefficients $\{A_{ijs} \mid A_{ij}(x) \neq 0, s = \deg A_{ij}(x), i < j\}$ is algebraically independent (over a subfield of* **F**), *provided* **F** *is not of characteristic two.*

Proof. The proof is immediate from the observation that a perfect matching M in G corresponds to a (nonzero) term a_P in the Pfaffian. □

It should be understood that $\delta(A) \leq \widehat{\delta}_1(A) \leq \widehat{\delta}_0(A)$ holds but that $\widehat{\delta}_0(A)$ can be strictly larger than $\widehat{\delta}_1(A)$ even in the generic case (see Example 4.1 below). Hence $\widehat{\delta}_0(A)$ is not suitable for the starting point of our algorithm, whereas $\widehat{\delta}_1(A)$ serves as the combinatorial relaxation of $\delta(A)$.

Our algorithm of combinatorial relaxation type, as outlined in §2.1, takes advantage of fast algorithms for nonbipartite matchings (see [1], [2], [8], [12], [16], [17], [19], [21], [31], [36]) to compute $\widehat{\delta}_1(A)$. Since the dual variables associated with a nonbipartite matching problem involve not only those for vertices but also those for odd subsets (or blossoms), Phase 2 and Phase 3 in the outlined algorithm demand substantial considerations not needed in §§2 and 3.

Remark 4.1. The algorithm for a skew-symmetric matrix, which will be developed in the rest of this paper, should be an extension of the algorithm of §2.4 for a general matrix. Given a square matrix A we may consider a skew-symmetric matrix $\bar{A} = \begin{pmatrix} O & A \\ -A^{\mathrm{T}} & O \end{pmatrix}$. Then $\delta(\bar{A}) = 2\delta(A)$ and $\widehat{\delta}_1(\bar{A}) = 2\widehat{\delta}_0(A)$. Thus, from the combinatorial point of view, the notion of skew-symmetric matrices is more general than that of general square matrices (!), as evidenced by the generalization of matroid to delta-matroid (see Bouchet [4], Chandrasekaran and Kabadi [7], and Dress and Havel [10]). □

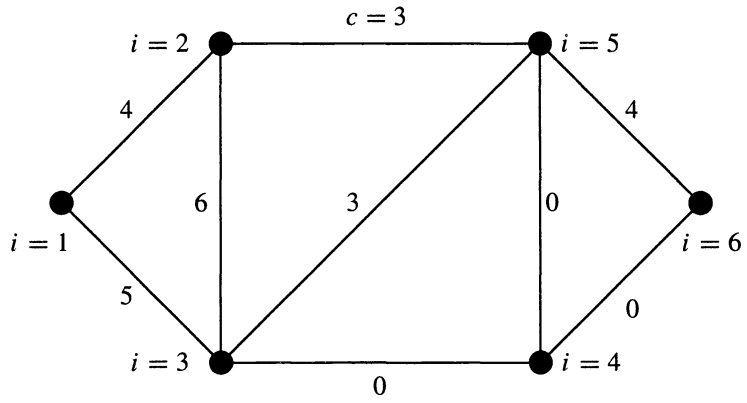


FIG. 3. *Graph* $G_1(A)$ (*Example* 4.1).

Example 4.1. Consider the following skew-symmetric matrix ($n = 6$):

$$A(x) = \begin{pmatrix} 0 & x^4+x & x^5 & 0 & 0 & 0 \\ -x^4-x & 0 & x^6 & 0 & x^3 & 0 \\ -x^5 & -x^6 & 0 & 1 & x^3 & 0 \\ 0 & 0 & -1 & 0 & 1 & \alpha \\ 0 & -x^3 & -x^3 & -1 & 0 & x^4 \\ 0 & 0 & 0 & -\alpha & -x^4 & 0 \end{pmatrix} \tag{4.2}$$

with a nonzero parameter α.

The associated graph $G = G_1(A)$, shown in Fig. 3, has 6 vertices and 9 edges. By inspection we see that it admits three perfect matchings with weights 8, 8, and 7, and

$$\mathrm{pf}A = A_{12}A_{34}A_{56} + A_{13}A_{25}A_{46} - A_{12}A_{35}A_{46} = (\alpha+1)x^8 - \alpha x^7 + x^5 - \alpha x^4.$$

Accordingly, we have

$$\widehat{\delta}_1(A) = 16, \quad \delta(A) = \begin{cases} 16 & \text{if } \alpha \neq -1, \\ 14 & \text{if } \alpha = -1. \end{cases}$$

Note also that $\widehat{\delta}_0(A) = 19$, which can never be equal to $\delta(A)$. □

4.2. Extraction of the tight part. In this section we must design an efficient procedure for Phase 2 by extending the idea used in §2.2. The concrete algorithm will be described in §4.4. Note that the upper tightness of a skew-symmetric A is defined with respect to $\widehat{\delta}_1(A)$ (and not to $\widehat{\delta}_0(A)$).

As an extension of (2.5) and (2.7) we consider the following primal-dual pair of linear programs. The PLP is a standard description of the perfect matching polytope.

$$\begin{aligned} &\text{PLP: maximize} && \sum_{e\in E} c_e\xi_e, \\ &\text{subject to} && \sum_{\partial e \ni i} \xi_e = 1 \quad (i \in V), \\ &&& \sum_{\partial e \subseteq S} \xi_e \leq \frac{|S|-1}{2} \quad (S \in \mathcal{S}), \\ &&& \xi_e \geq 0 \quad (e \in E); \end{aligned} \tag{4.3}$$

$$\text{DLP: minimize} \quad \sum_{i \in V} p_i + \sum_{S \in \mathcal{S}} \frac{|S|-1}{2} q_S \quad (\equiv \pi(p, q)),$$

$$\text{subject to} \quad p_i + p_j + \sum_{S \supseteq \{i,j\}} q_S \geq c_{ij} \quad ((i, j) \in E), \tag{4.4}$$

$$q_S \geq 0 \quad (S \in \mathcal{S}),$$

where

$$\mathcal{S} = \{S \subseteq V \mid |S| \geq 3,\ |S| \text{ is odd}\}. \tag{4.5}$$

Note that $\xi = (\xi_e \mid e \in E)$ is the primal variable and $(p, q) = (p_i \mid i \in V) \oplus (q_S \mid S \in \mathcal{S})$ is the dual variable.

With respect to q we define

$$\mathcal{S}^+ = \mathcal{S}^+(q) = \{S \in \mathcal{S} \mid q_S > 0\}, \tag{4.6}$$

which denotes the (index) set of the active dual variables q_S. With abuse of terminology, we call a member of $\mathcal{S}$ a *blossom* and one of $\mathcal{S}^+(q)$ an *active blossom* with respect to q. $\mathcal{S}^+$ is said to be *nested* if $S_1 \cap S_2 \neq \emptyset$ ($S_1 \in \mathcal{S}^+$, $S_2 \in \mathcal{S}^+$) implies either $S_1 \subseteq S_2$ or $S_1 \supseteq S_2$.

The (total dual half-) integrality stated below is crucial to our algorithm.[1]

PROPOSITION 4.3 (Edmonds [12]). (1) *PLP has an integral optimal solution (with* $\xi_e \in \{0, 1\}$ $(e \in E)$*).*

(2) *If* c_e *is integer for* $e \in E$*, DLP has an optimal solution* (p, q) *such that*

$$\text{(Int2)} \qquad p_i \in \frac{1}{2}\mathbf{Z} \quad (i \in V) \quad \text{and} \quad q_S \in \mathbf{Z} \quad (S \in \mathcal{S}), \tag{4.7}$$

$$\text{(Nest)} \qquad \mathcal{S}^+(q) \text{ is nested.} \qquad \square \tag{4.8}$$

By virtue of the primal integrality we have (cf. (2.7))

$$\widehat{\delta}_1(A) = 2 \min\{\pi(p, q) \mid (p, q) \text{ is feasible to DLP}\}. \tag{4.9}$$

Edmonds's primal-dual (blossom) algorithm [11], [12], [14] yields a dual optimal solution (p, q) that satisfies (Int2) and (Nest). Throughout this paper we assume (and maintain) dual variables satisfying the two conditions (Int2) and (Nest).

The optimality (complementarity) of a perfect matching, stated in Proposition 2.3 for bipartite matchings, is extended as follows. For $e = (i, j) \in E$, the reduced cost defined in (2.8) is modified to

$$\widetilde{c}_e = \widetilde{c}_{ij} = c_{ij} - p_i - p_j - Q_{ij}, \tag{4.10}$$

where

$$Q_{ij} = \sum\{q_S \mid \{i, j\} \subseteq S \in \mathcal{S}\}. \tag{4.11}$$

Then (p, q) is (dual) feasible if and only if $\widetilde{c}_e \leq 0$ $(e \in E)$ and $q_S \geq 0$ $(S \in \mathcal{S})$. An edge e is said to be tight (with respect to (p, q)) if $\widetilde{c}_e = 0$. We set

$$E^* = E^*(p, q) = \{e \in E \mid \widetilde{c}_e = 0\},$$

which is the set of tight edges. We also set $G^* = G^*(p, q) = (V, E^*(p, q))$.

[1] The total dual integrality, a stronger property than (Int2), is known (Cunningham and Marsh [8]) but half-integrality is sufficient and relevant for the subsequent argument.

PROPOSITION 4.4. *Let M be a perfect matching in $G(A)$ and (p, q) be a dual feasible solution. Then both M and (p, q) are optimal (i.e., $c(M) = \pi(p, q)$) if and only if $M \subseteq E^*(p, q)$ and $|\{e \in M \mid \partial e \subseteq S\}| = (|S| - 1)/2$ for all $S \in \mathcal{S}^+(q)$.* □

As in §2.2 we extract the "tight" part from $A(x)$, which is composed of the entries corresponding to the edges in E^*. For a dual feasible (p, q) we define

$$\mathcal{T}(A; p, q) = A^* = (A^*_{ij}), \quad A^*_{ij} = \begin{cases} A_{ij c_{ij}} & \text{if } (i, j) \in E^*(p, q), \\ 0 & \text{otherwise.} \end{cases} \tag{4.12}$$

That is,

$$A_{ij}(x) = x^{p_i + p_j + Q_{ij}}(A^*_{ij} + o(1)). \tag{4.13}$$

The "leveling" or "scaling" operation (2.11) is modified as

$$\mathcal{L}(A; p) = \operatorname{diag}(x; -p) \cdot A(x) \cdot \operatorname{diag}(x; -p). \tag{4.14}$$

PROPOSITION 4.5. *Let $\widetilde{A}(x) = \mathcal{L}(A; p)$.*

(1) $\delta(\widetilde{A}) = \delta(A) - 2\sum_{i \in V} p_i$, $\widehat{\delta}_1(\widetilde{A}) = \widehat{\delta}_1(A) - 2\sum_{i \in V} p_i$.

(2) *If (p, q) is dual feasible then $\widetilde{A}_{ij}(x) = x^{Q_{ij}}(A^*_{ij} + o(1))$.*

Proof. (1) The first relation is immediate from (4.14). The second (well-known) fact follows from the equality $\sum_{(i,j) \in M}(c_{ij} - p_i - p_j) = c(M) - \sum_{i \in V} p_i$, which holds true for any perfect matching M.

(2) This is a restatement of (4.13). □

Example 4.2 (continued from Example 4.1). We may take the following as the optimal dual variables:

$$p_1 = 0, \quad p_2 = 1, \quad p_3 = 2, \quad p_4 = -2, \quad p_5 = 2, \quad p_6 = 2;$$

$q_S = 3$ for $S = \{1, 2, 3\}$ and $q_S = 0$ otherwise. We have $\widehat{\delta}_1(A) = 2\pi(p, q) = 16$.

Those variables and the reduced costs are illustrated in Fig. 4. According to (4.14) and (4.12) we have

$$\widetilde{A}(x) = \mathcal{L}(A; p) = \begin{pmatrix} 0 & x^3 + 1 & x^3 & 0 & 0 & 0 \\ -x^3 - 1 & 0 & x^3 & 0 & 1 & 0 \\ -x^3 & -x^3 & 0 & 1 & x^{-1} & 0 \\ 0 & 0 & -1 & 0 & 1 & \alpha \\ 0 & -1 & -x^{-1} & -1 & 0 & 1 \\ 0 & 0 & 0 & -\alpha & -1 & 0 \end{pmatrix} \tag{4.15}$$

and

$$A^* = \mathcal{T}(A; p, q) = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 1 & 0 \\ -1 & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & \alpha \\ 0 & -1 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -\alpha & -1 & 0 \end{pmatrix}. \tag{4.16}$$

□

In analogy with Proposition 2.6 one might be tempted to claim that $A(x)$ is upper tight (i.e., $\delta(A) = \widehat{\delta}_1(A)$) if and only if A^* is nonsingular. The following example shows, however, that this is not the case.

Example 4.3. Consider the following skew-symmetric matrix ($n = 6$):

$$A(x) = \begin{pmatrix} 0 & x^3 & x^3 & x^2 & 0 & 0 \\ -x^3 & 0 & x^3 & 0 & \alpha x^2 & 0 \\ -x^3 & -x^3 & 0 & 0 & 0 & 1 \\ -x^2 & 0 & 0 & 0 & \beta x^2 & 0 \\ 0 & -\alpha x^2 & 0 & -\beta x^2 & 0 & 1 \\ 0 & 0 & -1 & 0 & -1 & 0 \end{pmatrix} \tag{4.17}$$

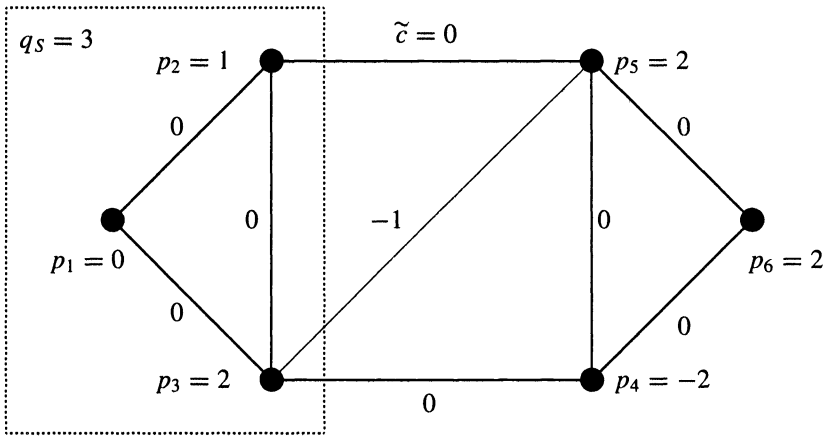


FIG. 4. *Reduced costs and dual variables for* $G_1(A)$ *(Example* 4.2).

with nonzero parameters α and β. As the optimal dual variables we may take $p_i = 0$ $(i = 1, \ldots, 6)$ and

$$q_S = \begin{cases} 1 & \text{for } S = S_1, \\ 2 & \text{for } S = S_2, \\ 0 & \text{otherwise,} \end{cases}$$

where $S_1 = \{1, 2, 3\}$ and $S_2 = \{1, 2, 3, 4, 5\}$. We have

$$A^* = \mathcal{T}(A; p, q) = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 & \alpha & 0 \\ -1 & -1 & 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 & \beta & 0 \\ 0 & -\alpha & 0 & -\beta & 0 & 1 \\ 0 & 0 & -1 & 0 & -1 & 0 \end{pmatrix}. \tag{4.18}$$

Direct expansions show that

$$\det A(x) = ((\beta + 1)x^5 - \alpha x^4)^2, \quad \det A^* = (\beta + 1 - \alpha)^2,$$

whereas $\widehat{\delta}_1(A) = 10$. With $\alpha = 2$ and $\beta = 1$ we see that the upper tightness of A does not imply $\det A^* \neq 0$, and with $\alpha = 2$ and $\beta = -1$ we see that the converse is not true either. Thus there is no relation between the upper tightness of A and the nonsingularity of A^*. □

To derive a necessary and sufficient condition for the upper tightness we extract the "tight" terms as follows. For a dual feasible (p, q) we define

$$\mathcal{U}(A; p, q) = A^\circ(x) = (A^\circ_{ij}(x)), \quad A^\circ_{ij}(x) = A^*_{ij} x^{p_i + p_j + Q_{ij}}. \tag{4.19}$$

Note that $A^* = A^\circ(1)$.

The following proposition states that in order to test for the upper tightness of $A(x)$ we can concentrate on the "tight" part $A^\circ(x)$. The last statement (4) says that an analogy of Proposition 2.6 holds true if $\mathcal{S}^+(q) = \emptyset$.

PROPOSITION 4.6. *Let* (p, q) *be a dual optimal solution for a skew-symmetric matrix* $A(x)$ *and* $A^\circ(x) = \mathcal{U}(A; p, q)$.

(1) $\widehat{\delta}_1(A) = \widehat{\delta}_1(A^\circ)$.

(2) *Let* β^* *denote the coefficient of* $x^{\widehat{\delta}_1(A)}$ *in* $\det A(x)$, *i.e.*,

$$\det A(x) = x^{\widehat{\delta}_1(A)}(\beta^* + o(1)).$$

Then

$$\det A^\circ(x) = x^{\widehat{\delta}_1(A)}(\beta^* + o(1)).$$

(3) $A(x)$ *is upper tight (i.e.,* $\delta(A) = \widehat{\delta}_1(A)$*) if and only if* $A^\circ(x)$ *is upper tight.*

(4) *Assume* $q = 0$ *and let* $A^* = \mathcal{T}(A; p, q)$. *Then* $\beta^* = \det A^*$ *and hence* $A(x)$ *is upper tight if and only if* A^* *is nonsingular.*

Proof. (1) By Proposition 4.4, $M \subseteq E$ is an optimal matching in G if and only if M is an optimal matching in G^*.

(2) By Proposition 4.1 we have $\det A = (\mathrm{pf}A)^2$, $\det A^\circ = (\mathrm{pf}A^\circ)^2$. Recall the definition of $\mathrm{pf}A$ (cf. §4.1), which is a sum of terms over all partitions or perfect matchings,

$$\mathrm{pf}A = \sum_{M:c(M)=\widehat{\delta}_1(A)/2} a_M + \sum_{M:c(M)<\widehat{\delta}_1(A)/2} a_M, \tag{4.20}$$

where $a_M = \pm \prod_{(i,j)\in M} A_{ij}$. The first summation yields the terms of degree $\widehat{\delta}_1(A)/2$ and the second yields those of lower degrees. Also, for $\mathrm{pf}A^\circ$ we have an expression similar to (4.20) consisting of two summations. The first summation on the right-hand side of (4.20) for A is taken over all optimal matchings in G, whereas the first summation for $\mathrm{pf}A^\circ$ is over all optimal matchings in G^*. This implies that those summations are identical, since the optimal matchings in G are nothing but the optimal matchings in G^* as noted above. Therefore we obtain

$$\mathrm{pf}A = \mathrm{pf}A^\circ + o(x^{\widehat{\delta}_1(A)/2}),$$

from which the claim follows immediately since $\det A = (\mathrm{pf}A)^2$ and $\det A^\circ = (\mathrm{pf}A^\circ)^2$.

(3) The upper tightness of A (as well as that of A°) is equivalent to $\beta^* \neq 0$.

(4) Since $q = 0$, Proposition 4.4 shows that M is an optimal matching in G if and only if M is a perfect matching in G^*. Hence the first summation on the right-hand side of (4.20) is equal to $x^{\widehat{\delta}_1(A)/2} \cdot \mathrm{pf}A^*$. Hence $\beta^* = (\mathrm{pf}A^*)^2 = \det A^*$. □

Proposition 4.6 above does not readily lead to an efficient algorithm for testing for the upper tightness since it does not provide a way to compute β^* when $\mathcal{S}^+(q) \neq \emptyset$. Based on the complementary slackness (Proposition 4.4), we will show in §4.4 that the test for upper tightness of $A(x)$ is reduced to the test for nonsingularity of a number of constant matrices derived from A^*. This makes it possible to determine efficiently (with $O(n^3)$ arithmetic operations in **F**) whether or not A is upper tight. First, in §4.3 we introduce a general modification scheme that will often be employed implicitly or explicitly in our algorithm.

4.3. General modification scheme. We prepare a general modification scheme, which will be used again and again in our algorithm. This modification scheme is nothing but a localized and symmetrized version of the operation introduced in §2.3.

Recall that $A^* = \mathcal{T}(A; p, q)$ is defined by (4.12) with respect to a dual feasible (p, q). Let $T \subseteq V$ be such that

$$A^*[T, T] \text{ is singular,} \tag{4.21}$$

where $A^*[T, T]$ means the submatrix of A^* with row and column indices in T, and

$$T \cap S \neq \emptyset,\ S \in \mathcal{S}^+(q) \quad \text{implies} \quad T \subseteq S. \tag{4.22}$$

The latter imposes a certain "locality" of T with respect to the active blossoms.

The modification of $A(x)$ to, say, $B(x)$ with respect to (T, p, q) is defined as follows.

GENERAL MODIFICATION SCHEME FOR (A, T, p, q).

Step 1: Since $A^*[T, T]$ is singular by assumption, there exists a nonzero vector $u = (u_i \in \mathbf{F} \mid i \in V)$ such that

$$\sum_{i \in T} u_i A^*_{ij} = 0 \ (j \in T), \quad \text{i.e.,} \quad u[T]^{\mathrm{T}} A^*[T, T] = 0, \tag{4.23}$$

where $u[T] = (u_i \mid i \in T)$. We choose u with minimal support; in particular, $\operatorname{supp} u \subseteq T$.

Step 2: Let $h \in \operatorname{supp} u$ be such that either

$$\text{(a): } p_h = \max\{p_i \mid i \in \operatorname{supp} u\} \text{ or (b): } p_h = \min\{p_i \mid i \in \operatorname{supp} u\}. \tag{4.24}$$

The index h thus determined is sometimes referred to as $\mathcal{H}(A; T, p, q)$.

Step 3: Divide u_i $(i \in V)$ by u_h (so that $u_h := 1$). The elimination matrix $U = (U_{ik} \mid i, k \in V)$ is defined by

$$U_{ik} := \begin{cases} u_k & \text{if } i = h, \\ \delta_{ik} & \text{otherwise.} \end{cases}$$

Step 4: The transformation matrix $U(x)$ is defined as

$$U(x) := \operatorname{diag}(x; p) \cdot U \cdot \operatorname{diag}(x; -p),$$

i.e.,

$$U_{ik}(x) := \begin{cases} x^{\sigma(h,k)} u_k & \text{if } i = h, \\ \delta_{ik} & \text{otherwise,} \end{cases} \tag{4.25}$$

where $\sigma(h, k) = p_h - p_k$.

Step 5: Finally, we put

$$B(x) := U(x) A(x) U(x)^{\mathrm{T}},$$

which will be referred to as $\mathcal{D}(A; T, p, q)$. Note that $\mathcal{L}(B; p) = U \cdot \mathcal{L}(A; p) \cdot U^{\mathrm{T}}$.

The properties of this modification scheme are summarized in Proposition 4.7 below.

PROPOSITION 4.7. *Suppose (p, q) is dual feasible with the property* (Int2) *of* (4.7). *If $T \subseteq V$ satisfies* (4.21) *and* (4.22) *then the following statements hold true:*

(1) $B(x) = -B(x)^{\mathrm{T}}$, $\delta(B) = \delta(A)$.

(2) $U(x) \in \mathcal{M}(\mathbf{F}[x, 1/x])$ *and hence* $B(x) \in \mathcal{M}(\mathbf{F}[x, 1/x])$. *If the criterion* (a) *is adopted in Step* 2, *then* $U(x) \in \mathcal{M}(\mathbf{F}[x])$.

(3) $Q_{ij} = Q_{kj}$ $(i, k \in T; j \in V)$, *where Q_{ij} is defined by* (4.11).

(4) (p, q) *is dual feasible for* B, *and hence* $\widehat{\delta}_1(B) \leq 2\pi(p, q)$.

(5) $B^* = \mathcal{T}(B; p, q)$ *satisfies* $B^*_{hj} = -B^*_{jh} = 0$ $(j \in T)$, *where* $h = \mathcal{H}(A; T, p, q)$.

(6) *If (p, q) is optimal for A, then* $\widehat{\delta}_1(B) \leq \widehat{\delta}_1(A)$.

Proof. (1) The first equality is obvious and the second follows from $\det U(x) = 1$.

(2) The matrix A^* is represented by the graph $G^* = G^*(p, q)$, which consists of the tight edges E^*. The submatrix $A^*[T, T]$ corresponds to the vertex-induced subgraph $G^*[T] = (T, E^*[T])$ of G^* induced on T, where $E^*[T] = \{(i, j) \in E^* \mid i \in T, j \in T\}$. Let C_1, $C_2, \ldots, (C_l \subseteq T)$ be the connected components of $G^*[T]$. Then the submatrix $A^*[T, T]$ is a block-diagonal matrix (or direct sum) with each diagonal block corresponding to a connected component $A^*[T, T] = \bigoplus_l A^*[C_l, C_l]$. Hence the minimality of $\operatorname{supp} u$ implies that $\operatorname{supp} u$

is contained in a single component. Therefore, if $i \in \operatorname{supp} u$ and $j \in \operatorname{supp} u$, the two vertices i and j are connected in $G^*[T]$ (and a fortiori in G^*).

On the other hand, we see that $p_i - p_j \in \mathbf{Z}$ for $(i, j) \in E^*$ since $p_i + p_j = c_{ij} - Q_{ij} \in \mathbf{Z}$ and $p_i \in \frac{1}{2}\mathbf{Z}$, $p_j \in \frac{1}{2}\mathbf{Z}$ by (Int2). Hence $p_i - p_j \in \mathbf{Z}$ if the two vertices $i \in V$ and $j \in V$ are connected in G^*. Hence we conclude that $p_i - p_j \in \mathbf{Z}$ if $i \in \operatorname{supp} u$ and $j \in \operatorname{supp} u$. In particular, $\sigma(h, k) = p_h - p_k \in \mathbf{Z}$ in the definition (4.25) of $U(x)$.

(3) Let $i \in T$ and $j \in V$, and note from (4.11) that $Q_{ij} = \sum\{q_S \mid \{i, j\} \subseteq S \in \mathcal{S}^+(q)\}$. If $\{i, j\} \subseteq S \in \mathcal{S}^+(q)$ then $T \cap S \neq \emptyset$, which implies $T \subseteq S$ by (4.22). Hence $Q_{ij} = \sum\{q_S \mid T \cup \{j\} \subseteq S \in \mathcal{S}^+(q)\}$, the right-hand side of which is independent of $i \in T$.

(4) Put $\widetilde{A}(x) = \mathcal{L}(A; p)$ and $\widetilde{B}(x) = \mathcal{L}(B; p)$. Then the dual feasibility of (p, q) for A is equivalent to $\deg \widetilde{A}_{ij} \leq Q_{ij}$ and that for B is equivalent to $\deg \widetilde{B}_{ij} \leq Q_{ij}$. The claim is obvious for (i, j) with $i, j \neq h$ since $\widetilde{B}_{ij} = \widetilde{A}_{ij}$. For $i = h$, $j \neq h$ we have $\widetilde{B}_{hj} = \sum_{i \in T} u_i \widetilde{A}_{ij}$ and $\deg \widetilde{A}_{ij} \leq Q_{ij} = Q_{hj}$ for $i \in T$ by (3). Hence $\deg \widetilde{B}_{hj} \leq Q_{hj}$. Thus (p, q) is feasible for B and (4.9) implies $\widehat{\delta}_1(B) \leq 2\pi(p, q)$.

(5) Substituting $\widetilde{A}_{ij}(x) = x^{Q_{ij}}(A^*_{ij} + o(1))$, $\widetilde{B}_{ij}(x) = x^{Q_{ij}}(B^*_{ij} + o(1))$ into $\widetilde{B}(x) = U\widetilde{A}(x)U^{\mathrm{T}}$ and noting (3) above, we obtain $B^* = UA^*U^{\mathrm{T}}$. Then the claim follows from (4.23).

(6) The optimality of (p, q) for A implies $\widehat{\delta}_1(A) = 2\pi(p, q)$, which is to be combined with the inequality in (4) to derive $\widehat{\delta}_1(B) \leq \widehat{\delta}_1(A)$. □

4.4. Test for upper tightness. Using the general modification scheme of §4.3, we now describe the algorithm for testing for the upper tightness of A. The algorithm is based on a matrix version of Edmonds's shrinking operation. It is assumed that (p, q) is optimal for A with the properties (Int2) and (Nest) of (4.7) and (4.8). Recall that the upper tightness of $A(x)$ is equivalent to that of $A^\circ(x) = \mathcal{U}(A; p, q)$ (cf. Proposition 4.6).

Before describing the algorithm in full generality, let us introduce the main idea by considering a simple case where $\mathcal{S}^+(q) = \{S_1, S_2\}$ with $S_1 \subset S_2$. Example 4.4 below will illustrate the following arguments. Set $B^{(0)}(x) = A(x)$, $B^{\circ(0)}(x) = \mathcal{U}(B^{(0)}; p, q)$, $(p^{(0)}, q^{(0)}) = (p, q)$.

First we deal with $S = S_1$. We apply the general modification scheme to $B^{\circ(0)}(x)$ with $T = T_1 = S_1$. Note that the prerequisite (4.21) is met since $|T_1|$ is odd and a skew-symmetric matrix of odd order is singular. Also note that the "locality" (4.22) is satisfied. Let

$$B^{(1)}(x) = \mathcal{D}(B^{\circ(0)}; T_1, p^{(0)}, q^{(0)}), \quad h_1 = \mathcal{H}(B^{\circ(0)}; T_1, p^{(0)}, q^{(0)})$$

and set $V_1 = T_1 - h_1$. By Proposition 4.7 the dual variable $(p^{(0)}, q^{(0)})$ remains feasible to $B^{(1)}(x)$. We modify $(p^{(0)}, q^{(0)})$ to $(p^{(1)}, q^{(1)})$ by

$$p_i^{(1)} = \begin{cases} p_i^{(0)} + q_{S_1}/2 & \text{for } i \in S_1 - h_1, \\ p_i^{(0)} & \text{otherwise,} \end{cases} \qquad q_S^{(1)} = \begin{cases} 0 & \text{for } S = S_1, \\ q_S^{(0)} & \text{otherwise.} \end{cases}$$

Since $B^{(1)}_{h_1 j}(x) = -B^{(1)}_{j h_1}(x) = 0$ $(j \in S_1 - h_1)$, the new dual variable $(p^{(1)}, q^{(1)})$ is feasible to $B^{(1)}$; we set $B^{\circ(1)}(x) = \mathcal{U}(B^{(1)}; p^{(1)}, q^{(1)})$. The dual objective value is invariant, i.e., $\pi(p^{(0)}, q^{(0)}) = \pi(p^{(1)}, q^{(1)})$. Thus we have eliminated S_1 from the set of blossoms, i.e., $\mathcal{S}^+(q^{(1)}) = \{S_2\}$. We have $B^{\circ(1)}_{ij}(x) = -B^{\circ(1)}_{ji}(x) = 0$ $(i \in V_1, j \in V - V_1)$ since $q_{S_1} > 0$.

Next we apply the similar procedure to $S = S_2$. Namely, we apply the general modification scheme to $B^{\circ(1)}(x)$ with $T = T_2 = (S_2 - S_1) \cup \{h_1\}$, where $|T_2|$ is odd. Note that "locality" (4.22) is satisfied for $q^{(1)}$ since $q^{(1)}_{S_1} = 0$. Let

$$B^{(2)}(x) = \mathcal{D}(B^{\circ(1)}; T_2, p^{(1)}, q^{(1)}), \quad h_2 = \mathcal{H}(B^{\circ(1)}; T_2, p^{(1)}, q^{(1)})$$

and set $V_2 = T_2 - h_2$. We modify the dual variable as

$$p_i^{(2)} = \begin{cases} p_i^{(1)} + q_{S_2}/2 & \text{for } i \in S_2 - h_2, \\ p_i^{(1)} & \text{otherwise,} \end{cases} \qquad q_S^{(2)} = \begin{cases} 0 & \text{for } S = S_2, \\ q_S^{(1)} & \text{otherwise.} \end{cases}$$

Since $B^{(2)}_{h_2 j}(x) = -B^{(2)}_{j h_2}(x) = 0$ $(j \in S_2 - h_2)$, $(p^{(2)}, q^{(2)})$ is feasible to $B^{(2)}$. We also have $\pi(p^{(1)}, q^{(1)}) = \pi(p^{(2)}, q^{(2)})$. Thus we have eliminated the blossoms, i.e., $\mathcal{S}^+(q^{(2)}) = \emptyset$.

Finally, we put $B(x) = B^{(2)}(x)$, $(p', q') = (p^{(2)}, q^{(2)})$, and $V_0 = V - (V_1 \cup V_2)$; note that V_0, V_1, and V_2 are pairwise disjoint. We may possibly have $h_1 = h_2$. It is very important that the final dual variable (p', q') has no blossoms by construction. Note also that $B^* = \mathcal{T}(B; p', q')$ takes a block-diagonal form, splitting into a direct sum

$$B^*[V_k, V_j] = O \quad (k \neq j;\ k, j = 0, 1, 2).$$

We are interested in the coefficient β^* of $x^{\widehat{\delta_1(A)}}$ in $\det A(x)$. If (p', q') remains optimal to B, then by Proposition 4.6 (2), β^* agrees with the coefficient of $x^{\delta_1(A)}$ in $\det B(x)$. Since $q' = 0$, we can apply Proposition 4.6 (4) to conclude

$$\beta^* = \det B^* = \det B^*[V_0, V_0] \cdot \det B^*[V_1, V_1] \cdot \det B^*[V_2, V_2].$$

This expression is valid even if (p', q') fails to be optimal to B (as a consequence of numerical cancellation) since both sides then vanish. This formula enables us to compute β^*, the nonvanishing of which is equivalent to the upper tightness of A.

Though we have derived the expression of β^* with reference to $B(x)$, we can compute β^* directly without involving the variable x and dual variable (p, q). Let us denote by C the matrix of coefficients of $B^\circ(x)$, i.e., $C = B^\circ(1)$; accordingly, we set $C^{(k)} = B^{\circ(k)}(1)$ $(k = 0, 1, 2)$. If we denote by U_1 and U_2 the (constant) transformation matrices found in the general modification scheme, i.e., $\mathcal{L}(B^{(1)}; p^{(0)}) = U_1 \cdot \mathcal{L}(B^{\circ(0)}; p^{(0)}) \cdot {U_1}^{\mathrm{T}}$, $\mathcal{L}(B^{(2)}; p^{(1)}) = U_2 \cdot \mathcal{L}(B^{\circ(1)}; p^{(1)}) \cdot {U_2}^{\mathrm{T}}$, we see that $C^{(k)}$ is obtained from $C^{(k-1)}$ by first making the product $\check{C}^{(k)} = U_k C^{(k-1)} {U_k}^{\mathrm{T}}$ and then suppressing the submatrices $\check{C}^{(k)}[V_{S_k}, V - V_{S_k}]$ and $\check{C}^{(k)}[V - V_{S_k}, V_{S_k}]$ to zeroes. Obviously, $C^{(2)} = B^*$ and, therefore,

$$\beta^* = \det C^{(2)}[V_0, V_0] \cdot \det C^{(2)}[V_1, V_1] \cdot \det C^{(2)}[V_2, V_2].$$

The operation for obtaining $B^* = C^{(2)}$ may be regarded as a matrix version of the shrinking operation in Edmonds's blossom algorithm. In particular, h_k $(k = 1, 2)$ corresponds to the bases of blossoms.

The basic idea illustrated above can be carried over to the general case by virtue of the property (Nest) of (4.8). The algorithm consists of alternate applications of the two distinct operations until $q = 0$ results:

(i) : Extract the tight part of B, i.e., $B^\circ(x) := \mathcal{U}(B; p, q)$.

(ii) : Apply the general modification scheme to B° to update B, i.e., $B(x) := \mathcal{D}(B^\circ; T, p, q)$, and adjust the dual variable (p, q).

Thus we obtain the following algorithm, which assumes that dual optimal solution (p, q) with properties (Int2) and (Nest) is given. It is emphasized here that the whole computation can be done with $O(n^3)$ arithmetic operations in $\mathbf{F}$. The implementation details to enhance the practical efficiency are left to the reader.

ALGORITHM FOR TESTING FOR THE UPPER TIGHTNESS OF A.

Step 1: $C := A^*$; $\overline{\mathcal{S}} := \mathcal{S}^+(q)$.

Step 2: If $\overline{\mathcal{S}} = \emptyset$ then go to Step 6, otherwise let S be a minimal element of $\overline{\mathcal{S}}$ and delete S from $\overline{\mathcal{S}}$;
$T := \left(S - \bigcup_{S' \in \mathcal{S}^+[S]} S'\right) \bigcup \{h_{S'} \mid S' \in \mathcal{S}^+[S]\}$,
where

$$\mathcal{S}^+[S] = \{S' \subset S \mid S' \text{ is a maximal member of } \mathcal{S}^+(q) \text{ properly contained in } S\}.$$

Step 3: Find a nonzero vector $u = (u_i \in \mathbf{F} \mid i \in V)$ such that

$$\sum_{i \in T} u_i C_{ij} = 0 \quad (j \in T),$$

where we choose u with minimal support; in particular $\operatorname{supp} u \subseteq T$.

Step 4: Choose $h \in \operatorname{supp} u$ arbitrarily and divide u_i $(i \in V)$ by u_h (so that $u_h := 1$). The elimination matrix $U = (U_{ik} \mid i, k \in V)$ is defined by

$$U_{ik} := \begin{cases} u_k & \text{if } i = h, \\ \delta_{ik} & \text{otherwise.} \end{cases}$$

Step 5: $h_S := h$; $V_S := T - h_S$;
$C := UCU^{\mathrm{T}}$; $C_{ij} := C_{ji} := 0$ $(i \in V_S, j \in V - V_S)$; Go to Step 2.

Step 6:

$$V_0 := V - \bigcup_{S \in \mathcal{S}^+(q)} V_S;$$

$$\beta^* := \det C[V_0, V_0] \cdot \prod_{S \in \mathcal{S}^+(q)} \det C[V_S, V_S];$$

If $\beta^* \neq 0$ then A is upper tight, otherwise A is not upper tight.

Thus we have obtained an efficient procedure for Phase 2, in which we must decide whether or not $\delta(A) = \widehat{\delta}_1(A)$. To sum up, Proposition 4.6, which is applicable only to the special case with $q = 0$, is now extended to the general case as follows.

PROPOSITION 4.8. *Let (p, q) be a dual optimal solution for a skew-symmetric matrix $A(x)$, and C be the matrix obtained from $A^* = \mathcal{T}(A; p, q)$ by the algorithm for testing the upper tightness of A.*

(1) $\det A(x) = x^{\widehat{\delta}_1(A)}(\beta^* + o(1))$, *where*

$$\beta^* = \det C[V_0, V_0] \cdot \prod_{S \in \mathcal{S}^+(q)} \det C[V_S, V_S].$$

(2) *$A(x)$ is upper tight (i.e., $\delta(A) = \widehat{\delta}_1(A)$) if and only if $\beta^* \neq 0$.*

Proof. The argument illustrated above for the case of $\mathcal{S}^+(q) = \{S_1, S_2\}$ with $S_1 \subset S_2$ carries over to the general case. To see this, consider another case where $\mathcal{S}^+(q) = \{S_1, S_2, S_3\}$ with $S_1 \cap S_2 = \emptyset$, $S_1 \subset S_3$, $S_2 \subset S_3$. Then the active blossoms are "shrunk" in the algorithm either in the order of indices S_1, S_2, S_3, or in the other order S_2, S_1, S_3. Since S_1 and S_2 are disjoint, the resulting coefficient matrix is the same. Thus the essential point is already captured in the simplest case. □

Example 4.4 (continued from Example 4.3). We apply the above arguments to the matrix $A(x)$ of (4.17) in Example 4.3. We have $\mathcal{S}^+(q) = \{S_1, S_2\}$, where $S_1 = \{1, 2, 3\}$ and $S_2 = \{1, 2, 3, 4, 5\}$. The matrix $B(x)$ and the dual variable (p, q) do not appear in the algorithm, but are important for understanding the procedure.

We have $B^{\circ(0)}(x) = B^{(0)}(x) = A(x)$ for this matrix. In Step 1 we initialize the matrix C to A^* of (4.18), and $\overline{\mathcal{S}} = \{S_1, S_2\}$. In Step 2 we have $S = S_1$, $\mathcal{S}^+[S] = \emptyset$, and $T = T_1 = S_1$. We may take $u^{\mathrm{T}} = (1, -1, 1, 0, 0, 0)$ in Step 3 and $h = h_1 = 3$ in Step 4. This means

$$B^{(1)}(x) = \begin{pmatrix} 0 & x^3 & 0 & x^2 & 0 & 0 \\ -x^3 & 0 & 0 & 0 & \alpha x^2 & 0 \\ 0 & 0 & 0 & x^2 & -\alpha x^2 & 1 \\ -x^2 & 0 & -x^2 & 0 & \beta x^2 & 0 \\ 0 & -\alpha x^2 & \alpha x^2 & -\beta x^2 & 0 & 1 \\ 0 & 0 & -1 & 0 & -1 & 0 \end{pmatrix}.$$

The dual variable is modified to

$$p_i^{(1)} = \begin{cases} \frac{1}{2} & \text{for } i = 1, 2, \\ 0 & \text{otherwise,} \end{cases} \qquad q_S^{(1)} = \begin{cases} 2 & \text{for } S = S_2, \\ 0 & \text{otherwise.} \end{cases}$$

We have $\pi(p^{(1)}, q^{(1)}) = 5 = \pi(p, q)$. The tight part of $B^{(1)}(x)$ with respect to the modified dual variable is given by

$$B^{\circ(1)}(x) = \mathcal{U}(B^{(1)}; p^{(1)}, q^{(1)}) = \begin{pmatrix} 0 & x^3 & 0 & 0 & 0 & 0 \\ -x^3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & x^2 & -\alpha x^2 & 1 \\ 0 & 0 & -x^2 & 0 & \beta x^2 & 0 \\ 0 & 0 & \alpha x^2 & -\beta x^2 & 0 & 1 \\ 0 & 0 & -1 & 0 & -1 & 0 \end{pmatrix}.$$

The matrix C is transformed to

$$C^{(1)} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -\alpha & 1 \\ 0 & 0 & -1 & 0 & \beta & 0 \\ 0 & 0 & \alpha & -\beta & 0 & 1 \\ 0 & 0 & -1 & 0 & -1 & 0 \end{pmatrix}$$

in Step 5.

Returning to Step 2 we obtain $S = S_2$, $\mathcal{S}^+[S] = \{S_1\}$, and $T = T_2 = (S_2 - S_1) \cup \{h_1\} = \{3, 4, 5\}$. Then we may take $u^{\mathrm{T}} = (0, 0, \beta, \alpha, 1, 0)$ in Step 3 and $h = h_2 = 5$ in Step 4. This means

$$B^{(2)}(x) = \begin{pmatrix} 0 & x^3 & 0 & 0 & 0 & 0 \\ -x^3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & x^2 & 0 & 1 \\ 0 & 0 & -x^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \beta + 1 \\ 0 & 0 & -1 & 0 & -\beta - 1 & 0 \end{pmatrix}.$$

The dual variable $(p^{(1)}, q^{(1)})$ is modified further to

$$p_i^{(2)} = \begin{cases} \frac{3}{2} & \text{for } i = 1, 2, \\ 1 & \text{for } i = 3, 4, \\ 0 & \text{otherwise,} \end{cases} \qquad q^{(2)} = 0.$$

Note that $\pi(p^{(2)}, q^{(2)}) = 5$. The matrix C is accordingly changed to

$$C^{(2)} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \beta+1 \\ 0 & 0 & 0 & 0 & -\beta-1 & 0 \end{pmatrix}.$$

In Step 6 we obtain $V_1 = \{1, 2\}$, $V_2 = \{3, 4\}$, $V_0 = \{5, 6\}$. The matrices $B(x)$ and C at the termination of the algorithm are, respectively, given by $B^{(2)}(x)$ and $C^{(2)}$. With respect to dual variable $(p', q') = (p^{(2)}, q^{(2)})$ we have $B^* = \mathcal{T}(B; p', q') = C^{(2)}$, which is in a block-diagonal form. Finally, we obtain

$$\begin{aligned} \beta^* &= \det B^* \\ &= \det B^*[V_1, V_1] \cdot \det B^*[V_2, V_2] \cdot \det B^*[V_0, V_0] \\ &= \det C[V_1, V_1] \cdot \det C[V_2, V_2] \cdot \det C[V_0, V_0] \\ &= \det \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \cdot \det \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \cdot \det \begin{pmatrix} 0 & \beta+1 \\ -\beta-1 & 0 \end{pmatrix} \\ &= (\beta+1)^2. \end{aligned}$$

As claimed in Proposition 4.8, this agrees with the coefficient of x^{10}, which has been obtained by direct expansion in Example 4.3.

The final matrix $B(x)$ is obtained from $A(x)$ as

$$B(x) = \mathcal{D}(\mathcal{U}(\mathcal{D}(\mathcal{U}(A); T_1)); T_2).$$

If we were to omit the operations $\mathcal{U}$ as

$$\overline{B}(x) = \mathcal{D}(\mathcal{D}(A; T_1); T_2),$$

we would obtain

$$\overline{B}(x) = \begin{pmatrix} 0 & x^3 & 0 & x^2 & \alpha x^2 & 0 \\ -x^3 & 0 & 0 & 0 & \alpha x^2 & 0 \\ 0 & 0 & 0 & x^2 & 0 & 1 \\ -x^2 & 0 & -x^2 & 0 & 0 & 0 \\ -\alpha x^2 & -\alpha x^2 & 0 & 0 & 0 & \beta+1 \\ 0 & 0 & -1 & 0 & -\beta-1 & 0 \end{pmatrix},$$

for which the final dual variable (p', q') is not feasible. This explains why the operation $\mathcal{U}$ has been introduced. Note also that, whereas $\overline{B}(x) = U(x)A(x)U(x)^{\mathrm{T}}$ for some unimodular $U(x)$, $B(x)$ is not obtained from $A(x)$ by such a unimodular congruence transformation. □

Remark 4.2. As explained for the simple situation, the algorithm above implicitly repeats extracting the tight part and transforming it by the general modification scheme. The resulting matrix has a dual optimal solution (p', q') with $q' = 0$. A possible alternative method is to resort to the procedure of "resolution of blossoms", which will be explained in §§4.5 and 4.6, and which transforms $A(x)$ to another matrix $\overline{B}(x) = U(x)A(x)U(x)^{\mathrm{T}}$ (where $\det U(x) = 1$), which admits a dual optimal solution (p'', q'') with $q'' = 0$. The latter approach, retaining all the nontight entries, is less efficient than the algorithm above. □

4.5. Transformation of a blossom-free matrix. When there is a gap between $\delta(A)$ and $\widehat{\delta}_1(A)$, the matrix $A(x)$ should be transformed to an upper-tight matrix through repeated unimodular congruence transformations (Phase 3).

The basic approach using the dual variables, as introduced in §§2 and 3 for the bipartite case of $\widehat{\delta}_0(A)$, also turns out to be effective for a skew-symmetric matrix. However, a novel technique must be devised before it can be implemented for the nonbipartite case, in which dual variables q_S associated with blossoms are involved. The technique, which we name the resolution of blossoms, bears resemblance to the operation of shrinking/expanding blossoms in Edmonds's algorithm.

Given a skew-symmetric $A(x) \in \mathcal{M}(\mathbf{F}[x, 1/x])$ with $\delta(A) < \widehat{\delta}_1(A)$, we will show how to modify $A(x)$ to another matrix $A'(x)$ such that

(P1-S) $A'(x) = U(x)A(x)U(x)^{\mathrm{T}}$ with $U(x) \in \mathcal{M}(\mathbf{F}[x, 1/x])$, $\det U(x) = 1$,

(P2-S) $\widehat{\delta}_1(A') \leq \widehat{\delta}_1(A) - 2$.

In addition, we sometimes impose either (P3a) or (P3b) of §2.3.

The algorithm to be proposed for this modification consists of the following two stages:

Resolving blossoms: Given a dual optimal solution (p, q) for A satisfying conditions (Int2) and (Nest) of (4.7) and (4.8), we find A' of (P1-S) and a dual feasible solution (p', q') for A' such that $q' = 0$, p' satisfies (Int2), and $\pi(p, q) = \pi(p', q')$. This implies that $\widehat{\delta}_1(A') \leq 2\pi(p', q') = 2\pi(p, q) = \widehat{\delta}_1(A)$. Since there are no blossoms (i.e., $\mathcal{S}^+(q') = \emptyset$) for the resulting matrix A', we call A' a *blossom-free matrix* and this operation the *resolution of blossoms*. At the end of this stage we set $A := A'$; $(p, q) := (p', q')$.

Decreasing gap for blossom-free matrix: Given a nontight A (with $\delta(A) < \widehat{\delta}_1(A)$), together with a dual optimal solution (p, q) satisfying (Int2) of (4.7) and $q = 0$, we find A' such that (P1-S) and (P2-S) are satisfied.

It is clear that alternate applications of these two stages result in an upper-tight matrix. We postpone the description of the stage of resolution of blossoms to §4.6 and consider the second stage here.

Let A be a nontight blossom-free matrix. Assume that $\delta(A) < \widehat{\delta}_1(A)$ and a dual optimal solution (p, q) satisfying (Int2) and $q = 0$ is available. Then $A^*[V, V]$ is singular by Proposition 4.6, and, therefore, the general modification scheme of §4.3 can be applied to A with $T = V$. The resulting matrix A' possesses the properties (P1-S) and (P2-S) as claimed below.

PROPOSITION 4.9. *Suppose a skew-symmetric $A(x) \in \mathcal{M}(\mathbf{F}[x, 1/x])$ satisfies $\delta(A) < \widehat{\delta}_1(A)$, and that (p, q) is a dual optimal solution satisfying* (Int2) *of* (4.7) *and $q = 0$. Then* (P1-S) *and* (P2-S) *are satisfied by $A'(x) = \mathcal{D}(A; V, p, q)$. In addition,* (P3a) *or* (P3b) *is met by the matrix $U(x)$ according to whether criterion* (a) *or* (b) *is used.*

Proof. Note that the "locality" condition (4.22) is trivially satisfied since $q = 0$. Then (P1-S) follows from Proposition 4.7.

Set $\widetilde{A}(x) = \mathcal{L}(A; p)$ and $\widetilde{B}(x) = \mathcal{L}(B; p)$, where $B(x) = A'(x) = \mathcal{D}(A; V, p, q)$. Note that (P2-S) is equivalent to $\widehat{\delta}_1(\widetilde{B}) \leq -2$ since $\widehat{\delta}_1(\widetilde{A}) = 0$ by $q = 0$. Using $\widetilde{A}_{ij}(x) = A^*_{ij} + o(1)$ (cf. Proposition 4.5 (2)) and (4.23) we obtain

$$\widetilde{B}_{hj}(x) = \sum_{i \in V} u_i \widetilde{A}_{ij}(x) = \sum_{i \in V} u_i A^*_{ij} + o(1) = o(1) \quad (j \neq h).$$

For (i, j) with $i, j \neq h$ we have $\widetilde{B}_{ij}(x) = \widetilde{A}_{ij}(x) = A^*_{ij} + o(1)$. Therefore $\widehat{\delta}_1(\widetilde{B}) < 0$, which implies $\widehat{\delta}_1(\widetilde{B}) \leq -2$ since $\widehat{\delta}_1(\widetilde{B}) \in 2\mathbf{Z}$ by (P1-S). □

Example 4.5 (continued from Example 4.2). Consider the following skew-symmetric matrix ($n = 6$):

$$A^{(1)}(x) = \begin{pmatrix} 0 & x^4 + x & -x^2 & 0 & 0 & 0 \\ -x^4 - x & 0 & -x^3 & 0 & x^3 & 0 \\ x^2 & x^3 & 0 & 1 & -x^4 + x^3 & 0 \\ 0 & 0 & -1 & 0 & 1 & \alpha \\ 0 & -x^3 & x^4 - x^3 & -1 & 0 & x^4 \\ 0 & 0 & 0 & -\alpha & -x^4 & 0 \end{pmatrix} \tag{4.26}$$

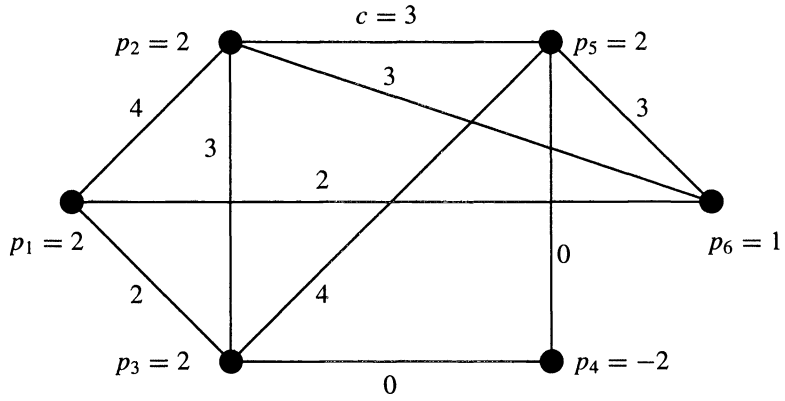


FIG. 5. *Graph $G_1(A)$ with modified $A = A^{(2)}$ (Example 4.5).*

with α being a nonzero parameter. It may be mentioned that this matrix is obtained from $A(x)$ of (4.2) in Example 4.1 by the resolution of blossoms (to be described in §4.6), though this fact is not used here.

The dual variables $(p, q) = (p^{(0)}, q^{(0)})$ with $q^{(0)} \neq 0$ given in Example 4.2 are also optimal for $A^{(1)}$. The matrix $A^{(1)}$, however, admits an alternative set of optimal dual variables $(p, q) = (p^{(1)}, q^{(1)})$ with $q^{(1)} = 0$ and

$$p_1^{(1)} = \tfrac{3}{2}, \quad p_2^{(1)} = \tfrac{5}{2}, \quad p_3^{(1)} = 2, \quad p_4^{(1)} = -2, \quad p_5^{(1)} = 2, \quad p_6^{(1)} = 2,$$

and hence $A^{(1)}$ is blossom free. Using the blossom-free dual variable $(p^{(1)}, q^{(1)})$ we transform $A^{(1)}$ according to the proposed procedure. We have

$$A^* = \mathcal{T}(A^{(1)}; p^{(1)}, q^{(1)}) = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & -1 & 0 & 1 & \alpha \\ 0 & 0 & 1 & -1 & 0 & 1 \\ 0 & 0 & 0 & -\alpha & -1 & 0 \end{pmatrix}.$$

Since $\det A^* = (\alpha + 1)^2$, let us assume $\alpha = -1$ for $A^{(1)}(x)$ to be nontight (cf. Proposition 4.6). The vector u of (4.23) is given by $u^{\mathrm{T}} = (0, 0, -1, 0, 0, 1)$, $\operatorname{supp} u = \{3, 6\}$, and $h = 6$ according to either criterion (a) or (b). We obtain $A^{(2)} = \mathcal{D}(A^{(1)}; V, p^{(1)}, q^{(1)})$:

$$A^{(2)}(x) = \begin{pmatrix} 0 & x^4 + x & -x^2 & 0 & 0 & x^2 \\ -x^4 - x & 0 & -x^3 & 0 & x^3 & x^3 \\ x^2 & x^3 & 0 & 1 & -x^4 + x^3 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & -x^3 & x^4 - x^3 & -1 & 0 & x^3 \\ -x^2 & -x^3 & 0 & 0 & -x^3 & 0 \end{pmatrix},$$

for which we find an (integer-valued) optimal dual variable $(p, q) = (p^{(2)}, q^{(2)})$:

$$p_1^{(2)} = 2, \quad p_2^{(2)} = 2, \quad p_3^{(2)} = 2, \quad p_4^{(2)} = -2, \quad p_5^{(2)} = 2, \quad p_6^{(2)} = 1;$$

and $q^{(2)} = 0$.

The associated graph $G_1(A^{(2)})$ is depicted in Fig. 5, which should be compared with $G_1(A)$ of Fig. 3. We see that $A^{(2)}$ is upper tight with $\delta(A^{(2)}) = \widehat{\delta}_1(A^{(2)}) = 2\pi(p^{(2)}, q^{(2)}) = 14$ since

$$A^* = \mathcal{T}(A^{(2)}; p^{(2)}, q^{(2)}) = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & -1 & 0 & 1 \\ 0 & -1 & 0 & 0 & -1 & 0 \end{pmatrix}$$

is nonsingular and $q^{(2)} = 0$. □

4.6. Resolution of blossoms. We present an algorithm for the resolution of blossoms, which is introduced in §4.5 as a sort of preprocessing operation in Phase 3. In testing for the upper tightness of $A(x)$, we have already shown in §4.4 how to derive from $A(x)$ another matrix $B(x)$ which admits an optimal dual solution (p, q) with $q = 0$. The algorithm to be presented here is a generalization of the same idea.

Given a skew-symmetric $A(x) \in \mathcal{M}(\mathbf{F}[x, 1/x])$ along with a dual optimal solution (p, q) satisfying the conditions (Int2) and (Nest) of (4.7) and (4.8), we will find $U(x) \in \mathcal{M}(\mathbf{F}[x, 1/x])$ with $\det U(x) = 1$ (cf. (P1-S)) such that $A'(x) = U(x)A(x)U(x)^{\mathrm{T}}$ admits a dual feasible solution (p', q'), which is blossom free ($q' = 0$), half-integral (p' satisfying (Int2)), and has the same objective value ($\pi(p, q) = \pi(p', q')$). This implies that $\widehat{\delta}_1(A') \leq 2\pi(p', q') = 2\pi(p, q) = \widehat{\delta}_1(A)$. The resolution of blossoms can be applied irrespective of the upper tightness of A (i.e., $\delta(A) < \widehat{\delta}_1(A)$ is not assumed here).

Our algorithm repeats resolving a minimal member of $\mathcal{S}^+(q)$ by modifying A and (p, q).

PROPOSITION 4.10. *Let (p, q) be a dual optimal solution for A satisfying the condition* (Nest) *of* (4.8)*, and S be a minimal member of $\mathcal{S}^+(q)$.*

(1) *$A^*[S, S]$ is singular, i.e.,* (4.21) *is satisfied with $T = S$.*

(2) *The "locality" condition* (4.22) *is satisfied with $T = S$.*

Proof. (1) $|S|$ is odd and a skew-symmetric matrix of odd order is singular.

(2) The proof is obvious from the minimality of S. □

A minimal blossom S is eliminated by repeatedly changing the matrix A with the general modification scheme of §4.3 and adapting the dual variables. Note that Proposition 4.10 shows that the modification scheme in §4.3 is applicable with $T = S$. Recall the notations $\mathcal{D}(A; T, p, q)$ and $\mathcal{H}(A; T, p, q)$.

ALGORITHM FOR THE RESOLUTION OF A MINIMAL BLOSSOM S.

Step 1: $B := A$; $(\bar{p}, \bar{q}) := (p, q)$.

Step 2: Apply the general modification scheme to $(B, S, \bar{p}, \bar{q})$ to obtain $B := \mathcal{D}(B; S, \bar{p}, \bar{q})$ and $h := \mathcal{H}(B; S, \bar{p}, \bar{q})$.

Step 3: $\epsilon_1 := \min_{j \in S-h}\{-\deg B_{hj}(x) + \bar{p}_h + \bar{p}_j + \bar{Q}_{hj}\}$, where $\bar{Q}_{hj}$ is defined by (4.11) with reference to $\bar{q}$;
$\epsilon_2 := \bar{q}_S/2$;
If $\epsilon_1 < \epsilon_2$ then go to Step 4, otherwise go to Step 5.

Step 4: $\bar{p}_i := \bar{p}_i + \epsilon_1 \quad (i \in S - h)$; $\bar{q}_S := \bar{q}_S - 2\epsilon_1$; Go to Step 2.

Step 5: $\bar{p}_i := \bar{p}_i + \epsilon_2 \quad (i \in S - h)$; $\bar{q}_S := \bar{q}_S - 2\epsilon_2 \ (= 0)$; Stop.

The behaviors of the above algorithm are stated below.

PROPOSITION 4.11. *Let (p, q) be a dual optimal solution satisfying* (Int2) *and* (Nest) *of* (4.7) *and* (4.8)*, and S be a minimal member of $\mathcal{S}^+(q)$. Referring to the above algorithm we have the following statements:*

(1) *At each execution of Step* 3*, we have $\epsilon_1 \in \frac{1}{2}\mathbf{Z}$, $\epsilon_1 \geq \frac{1}{2}$. Hence, at the beginning of Step* 2*, $(\bar{p}, \bar{q})$ satisfies the condition* (Int2) *of* (4.7).

(2) *The algorithm terminates after at most q_S executions of Step* 4.

Let B, h, $(\bar{p}, \bar{q})$ denote the variables at the termination of the algorithm.

(3) *$B(x) = U(x)A(x)U(x)^{\mathrm{T}}$ with $U(x) \in \mathcal{M}(\mathbf{F}[x, 1/x])$, $\det U(x) = 1$, and, therefore, $B(x) = -B(x)^{\mathrm{T}}$, $\delta(B) = \delta(A)$, and $B(x) \in \mathcal{M}(\mathbf{F}[x, 1/x])$. If criterion* (a) *is adopted consistently in the modification scheme employed in Step* 2*, then $U(x) \in \mathcal{M}(\mathbf{F}[x])$.*

(4) *$\bar{q}_S = 0$, $\bar{q}_{S'} = q_{S'}$ $(S \neq S' \in \mathcal{S}^+(q))$, and $\bar{p}_i \in \frac{1}{2}\mathbf{Z}$ $(i \in V)$, $\bar{p}_i = p_i$ $(i \in V - S)$.*

(5) *$\pi(\bar{p}, \bar{q}) = \pi(p, q)$.*

(6) *$(\bar{p}, \bar{q})$ is feasible to $B(x)$ and hence $\widehat{\delta}_1(B) \leq 2\pi(\bar{p}, \bar{q}) = 2\pi(p, q) = \widehat{\delta}_1(A)$.*

Proof. During the iterations of Steps 2–4, we have $\deg B_{hj} \in \mathbf{Z}$ and $\epsilon_1 \in \frac{1}{2}\mathbf{Z}$. Hence (1) and (2) are established. Then (3) follows from Proposition 4.7. In Step 5 we have $\epsilon_2 \in \frac{1}{2}\mathbf{Z}$,

which, combined with (1), implies the claims of (4). In the updates of $(\bar{p}, \bar{q})$ in Steps 4–5, $\pi(\bar{p}, \bar{q})$ remains unchanged since $|S - h| = 2 \times ((|S| - 1)/2)$. The feasibility of $(\bar{p}, \bar{q})$ to $B(x)$ claimed in (6) follows from Proposition 4.7. □

The algorithm for the resolution of all blossoms is now presented. It is assumed that (p, q) is a dual optimal solution satisfying the conditions (Int2) and (Nest) for a skew-symmetric $A(x) \in \mathcal{M}(\mathbf{F}[x, 1/x])$. Throughout the following algorithm $\mathcal{S}_0^+$ means $\mathcal{S}^+(q)$ for the given q, i.e., $\mathcal{S}_0^+ \equiv \mathcal{S}^+(q)$. For $S \in \mathcal{S}_0^+$ we define

$$\mathcal{S}_0^+[S] = \{S' \subset S \mid S' \text{ is a maximal member of } \mathcal{S}_0^+ \text{ properly contained in } S\}.$$

The essence of the algorithm below is to apply the resolution algorithm for a minimal blossom to all the members $S \in \mathcal{S}_0^+$ in an order which is consistent with the partial order defined by set inclusion. Proposition 4.12 below shows that the relevant conditions are satisfied by S.

ALGORITHM FOR THE RESOLUTION OF BLOSSOMS.

Step 1: $B := A$; $(\bar{p}, \bar{q}) := (p, q)$; $\mathcal{S}_0^+ := \mathcal{S}^+(q)$.

Step 2: If $\mathcal{S}^+(\bar{q}) = \emptyset$ then stop, otherwise let S be a minimal element of $\mathcal{S}^+(\bar{q})$.

Step 3: Apply the general modification scheme to $(B, S, \bar{p}, \bar{q})$ to obtain $B := \mathcal{D}(B; S, \bar{p}, \bar{q})$ and $h := \mathcal{H}(B; S, \bar{p}, \bar{q})$.

Step 4: $\epsilon_1 := \min_{j \in S-h}\{-\deg B_{hj}(x) + \bar{p}_h + \bar{p}_j + \bar{Q}_{hj}\}$, where $\bar{Q}_{hj}$ is defined by (4.11) with reference to $\bar{q}$;
$\epsilon_2 := \bar{q}_S/2$;
If $\epsilon_1 < \epsilon_2$ then go to Step 5, otherwise go to Step 6.

Step 5: $\bar{p}_i := \bar{p}_i + \epsilon_1 \quad (i \in S - h)$; $\bar{q}_S := \bar{q}_S - 2\epsilon_1$; Go to Step 3.

Step 6: $\bar{p}_i := \bar{p}_i + \epsilon_2 \quad (i \in S - h)$; $\bar{q}_S := \bar{q}_S - 2\epsilon_2 \ (= 0)$;
$T := \left(S - \bigcup_{S' \in \mathcal{S}_0^+[S]} S'\right) \bigcup \{h_{S'} \mid S' \in \mathcal{S}_0^+[S]\}$;
$h_S := h$; $V_S := T - h_S$; Go to Step 2. □

PROPOSITION 4.12. *For Step* 3 *of the above algorithm we have the following statements:*
(1) $B^*[S, S]$, *where* $B^* = \mathcal{T}(B; \bar{p}, \bar{q})$, *is singular, i.e.,* (4.21) *is satisfied.*
(2) *The "locality" condition* (4.22) *is satisfied with respect to* $T = S$ *and* $q = \bar{q}$.
(3) *The condition* (Int2) *of* (4.7) *with respect to* $(\bar{p}, \bar{q})$ *is satisfied.*

Proof. (1) $|S|$ is odd and a skew-symmetric matrix of odd order is singular.
(2) This holds because S is a minimal element of $\mathcal{S}^+(\bar{q})$.
(3) The proof is the same as the proof for Proposition 4.11(4). □

It should be clear that $\mathcal{S}^+(\bar{q})$ decreases monotonically starting with $\mathcal{S}_0^+$, which is bounded by $n/2$ in size. Hence Step 2 is executed at most $n/2$ times. On the other hand, Proposition 4.11 bounds the number of loops of Step 3 to Step 5. Namely, Step 3 is executed at most $\sum_S (q_S + 1)$ times. Therefore the above algorithm terminates.

For completeness we summarize the properties of the matrix $B(x)$ and the dual variable $(\bar{p}, \bar{q})$ at the termination of the above algorithm, though they follow readily from the statements in Proposition 4.11.

PROPOSITION 4.13. *Let* (p, q) *be a dual optimal solution satisfying* (Int2) *and* (Nest) *of* (4.7) *and* (4.8) *for a skew-symmetric* $A(x) \in \mathcal{M}(\mathbf{F}[x, 1/x])$. *Let* $B(x)$ *and* $(\bar{p}, \bar{q})$ *be the variables at the termination of the algorithm for the resolution of all blossoms.*

(1) $B(x) = U(x)A(x)U(x)^{\mathrm{T}}$ *with* $U(x) \in \mathcal{M}(\mathbf{F}[x, 1/x])$, $\det U(x) = 1$, *and, therefore,* $B(x) = -B(x)^{\mathrm{T}}$, $\delta(B) = \delta(A)$, *and* $B(x) \in \mathcal{M}(\mathbf{F}[x, 1/x])$. *If criterion* (a) *is adopted consistently in the modification scheme employed in Step* 3, *then* $U(x) \in \mathcal{M}(\mathbf{F}[x])$.

(2) $(\bar{p}, \bar{q})$ *is a feasible solution to* $B(x)$ *such that* $\bar{q} = 0$, $(\bar{p}, \bar{q})$ *satisfies* (Int2), *and* $\pi(\bar{p}, \bar{q}) = \pi(p, q)$. *Hence* $\widehat{\delta}_1(B) \leq 2\pi(\bar{p}, \bar{q}) = 2\pi(p, q) = \widehat{\delta}_1(A)$. □

Example 4.6 (continued from Example 4.5). As mentioned in Example 4.5, the matrix $A^{(1)}$, as well as $(p^{(1)}, q^{(1)})$, is obtained from $A(x)$ of (4.2) in Example 4.1 by the resolution of blossoms. We see that $h = 3$ (according to criterion (a)) and $A^{(1)}(x) = U(x)A(x)U(x)^{\mathrm{T}}$ with

$$U(x) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ x^2 & -x & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}. \qquad \square$$

4.7. Description of algorithm. We summarize here the whole algorithm for computing $\delta(A)$ for a skew-symmetric $A(x) \in \mathcal{M}(\mathbf{F}[x, 1/x])$.

ALGORITHM FOR COMPUTING $\delta(A)$ (A: SKEW-SYMMETRIC MATRIX).

Step 0

$c_{\min} := \min_{i,j} \min\{s \mid A_{ijs} \neq 0\}$.

Step 1

(1): Find a maximum weight perfect matching M in $G_1(A)$ and a dual optimal solution (p, q) satisfying the conditions (Int2) and (Nest) of (4.7) and (4.8); $\widehat{\delta}_1(A) := 2 \cdot c(M)$ ($\widehat{\delta}_1(A) = -\infty$ if no perfect matching exists).

(2): If $\widehat{\delta}_1(A) < n \cdot c_{\min}$, then stop with $\delta(A) := -\infty$.

Step 2

(1): $A^*_{ij} :=$ coefficient of $x^{p_i+p_j+Q_{ij}}$ in $A_{ij}(x)$ $(i, j \in V)$, where

$$Q_{ij} = \sum\{q_S \mid \{i, j\} \subseteq S \in \mathcal{S}\}.$$

(2): Test for the upper tightness of A by applying the algorithm of §4.4 to A^*. [cf. Prop.4.8]

(3): If A is upper tight, then stop with $\delta(A) := \widehat{\delta}_1(A)$.

Step 3 [$\delta(A) \neq \widehat{\delta}_1(A)$]

(1): Apply to (A, p, q) the algorithm for the resolution of blossoms to obtain $(B, \bar{p}, \bar{q})$ with $\bar{q} = 0$. [cf. §4.6]

(2): If $(\bar{p}, \bar{q})$ is optimal to B then go to Step 4, otherwise go to Step 5.

Step 4 [Decrease gap between δ and $\widehat{\delta}_1$]

(1): $A := B$; $(p, q) := (\bar{p}, \bar{q})$.

(2): Apply the general modification scheme to (A, V, p, q) to obtain $B := \mathcal{D}(A, V, p, q)$. [cf. §4.3]

Step 5 [cf. $\widehat{\delta}_1(B) \leq \widehat{\delta}_1(A) - 2$]

$A := B$; Go to Step 1.

Note in Step 3 (2) that $(\bar{p}, \bar{q})$ (with $\bar{q} = 0$) is optimal to B if and only if there exists a perfect matching in the tight part $G^*(\bar{p}, \bar{q})$ of $G(B)$ (cf. Proposition 4.4).

Since this algorithm (with criterion (a) in Step 2 of the general modification scheme) terminates after a finite number of steps, it gives a combinatorial proof for the following existence theorem [27], which is similar to Proposition 3.3.

PROPOSITION 4.14. *For a nonsingular skew-symmetric polynomial matrix* $A(x) \in \mathcal{M}(\mathbf{F}[x])$, *there exists a unimodular matrix* $U(x) \in \mathcal{M}(\mathbf{F}[x])$ *such that* $A'(x) = U(x)A(x)U(x)^{\mathrm{T}}$ *is upper tight, i.e.,* $(\delta(A) =)\ \delta(A') = \widehat{\delta}_1(A')$. $\square$

The complexity issues will be discussed in §5.

5. Complexity. In this section we discuss the computational complexity of the algorithm of §4.7 for a skew-symmetric polynomial matrix. Similar arguments apply to the other two algorithms of §§2.4 and 3.3 mutatis mutandis, though they are not expounded in detail.

The finite termination of the algorithm is guaranteed since $\widehat{\delta}_1(A)$ decreases at least by two in Step 5, and each step (in particular, the resolution of blossoms in Step 3) terminates in a finite number of steps as discussed in §4.6. To be more specific, Step 1 is executed at most $nc_{\max}/2$ times, where

$$c_{\max} = \max_{i,j} \deg A^{(0)}_{ij}(x)$$

denotes the maximum degree of an entry of the input matrix $A^{(0)}$.

In Step 1, the optimal matching, as well as the optimal dual solution, can be found in $O(n^3)$ time, or more efficiently (see [2], [8], [12], [16], [17], [21], [31]). The upper-tightness testing in Step 2 can be done with $O(n^3)$ arithmetic operations in $\mathbf{F}$ by means of a variant of the Gaussian elimination. In the case where A is upper tight, we stop at this point without any modification operations on A. In case A should turn out to be nontight, because of an accidental numerical cancellation, we are ready to go on to Phase 3, which consists of Steps 3–5.

As has been emphasized several times, the proposed algorithm should enjoy the average-case efficiency since the "lucky" case is usually executed where the algorithm terminates in Step 2. As before, let us fix the structure (i.e., the index set $\{(i, j, s) \mid A_{ijs} \neq 0\}$ of nonzero coefficients in (2.1)) of the input matrix $A = A^{(0)}$ and assume that the numerical values of coefficients $A_{ijs} \in \mathbf{R} = \mathbf{F}$ $(i < j)$ can be modeled as real-valued random variables with continuous distributions. Then the average time complexity (in this sense) of the proposed algorithm for a skew-symmetric matrix is bounded by a polynomial in n (e.g., n^3).

Let us now turn to the worst-case complexity. Step 4 of the main procedure, as well as the resolution of blossoms in Step 3, relies on the general modification scheme of §4.3, the complexity of which governs the overall complexity of the algorithm. We assume that criterion (b) is always adopted in Step 2 of the general modification scheme. A crucial observation follows.

PROPOSITION 5.1. *Suppose that criterion* (b) *is consistently adopted in Step* 2 *of the general modification scheme. Then we always have*

$$\max_{i,j} \deg A_{ij}(x) \leq c_{\max}$$

for the matrix A, which changes during the computation.

Proof. The definition of $B = \mathcal{D}(A; T, p, q)$ in Step 5 of the general modification scheme can be written componentwise as

$$B_{hj}(x) = \sum_{k \in T} x^{\sigma(h,k)} u_k A_{kj}(x) = \sum_{k \in T} \sum_{s \in \mathbf{Z}} u_k A_{kjs} x^{\sigma(h,k)+s}, \tag{5.1}$$

where $j \neq h$. Criterion (b) implies $\sigma(h, k) \leq 0$ and, therefore,

$$\max_{i,j} \deg B_{ij}(x) \leq \max_{i,j} \deg A_{ij}(x).$$

Hence the claim follows. □

It is remarked here that the dual optimal solutions (p, q) for A can be chosen in such a way that they are polynomially bounded by $(n, c_{\max})$.

Steps 1 to 4 of the general modification scheme can be done with $O(n^3)$ arithmetic operations in $\mathbf{F}$; note in particular that the vector u with minimal support in Step 1 can be found by a variant of the Gaussian elimination algorithm. The computation of B in Step 5 can be written componentwise as (5.1). This shows that the required amount of computation depends on the number of terms in $A_{kj}(x)$. Let us assume that the number of nonzero terms in $A_{kj}(x)$ is bounded, say by N. Then the general modification scheme can be done with $O(n^3 + Nn^2)$ arithmetic operations in $\mathbf{F}$.

Next we consider the complexity of the resolution of blossoms, which is called for in Step 3 (1) of the main procedure. As we mentioned in §4.6 (see also Proposition 4.11), the resolution of blossoms involves at most $\sum_S (q_S + 1)$ applications of the general modification scheme. Hence the complexity of the resolution of blossoms is bounded by $O((n + N)n^2 \sum_S q_S)$, which is polynomially bounded by $(n, c_{\max}, N)$ since, as remarked above, q is polynomially bounded by $(n, c_{\max})$.

As for Step 3 (2) of the main procedure, we have already noted in §4.7 that the optimality of $(\bar{p}, \bar{q})$ (with $\bar{q} = 0$) to B is equivalent to the existence of a perfect matching in the tight part $G^*(\bar{p}, \bar{q})$ of $G(B)$, and the latter can be checked in $O(n^3)$ time or less by means of the (unweighted) matching algorithm.

To sum up, the worst-case complexity of the entire algorithm is bounded by a polynomial in $(n, c_{\max}, N)$. This bound involves an upper bound N on the number of terms in $A_{ij}(x)$, which changes during the computation.

To establish a complexity bound in terms of $(n, c_{\max})$ only, we will show that it suffices to retain a polynomial number of terms for each entry of A. A similar idea was used by Murota [26]. Then, obviously, the number of terms in each entry of $A^{(0)}$ is bounded by $c_{\max}$, and we always have

$$0 \leq \delta(A^{(0)}) = \delta(A) \leq \widehat{\delta}_1(A) \leq \widehat{\delta}_1(A^{(0)}) \leq n \cdot c_{\max} \tag{5.2}$$

provided that $\delta(A^{(0)}) \neq -\infty$.

On the basis of Proposition 5.1 and (5.2) it is easy to see that a term of $A_{ij}(x)$ with degree less than $-n \cdot c_{\max}$ can be discarded without any influence on $\delta(A)$ or $\widehat{\delta}_1(A)$. In other words, we retain those terms of $A_{ij}(x)$ having degree $\geq -n \cdot c_{\max}$ (and $\leq c_{\max}$). This shows that we may assume $N \leq (n + 1)c_{\max}$.

PROPOSITION 5.2. *The proposed algorithm for a skew-symmetric polynomial matrix can be implemented to run in time polynomial in n and $c_{\max}$, where it is assumed that an arithmetic operation in* $\mathbf{F}$ *can be done in a constant time, and* $N \leq (n + 1)c_{\max}$ *for the input matrix.* □

6. Conclusion. This paper has presented "combinatorial relaxation-" type algorithms for computing the degree of determinant $\delta(A)$. These algorithms have the distinguished feature that they are based on the generic characterizations of $\delta(A)$ and enjoy the average-case efficiency. The combinatorial relaxation approach can be extended to computing the maximum degree of subdeterminants of a specified order, and hence to computing the Smith–McMillan form at infinity of polynomial/rational matrices and the structural indices in the Kronecker form of matrix pencils as reported in Murota [29].

For the computation of $\delta(A)$ there are a number of different algorithms available. A standard method would be the interpolation method that substitutes numerical values, say, α_i $(i = 1, \ldots, K)$ for variable x, computes the determinants $f(\alpha_i) = \det A(\alpha_i)$ $(i = 1, \ldots, K)$ by Gaussian elimination, and then constructs $f(x) = \sum_s f_s x^s$ by interpolation. This method is superior in theoretical complexity [3] to the combinatorial relaxation method. (However, it

is not clear how to extend the interpolation method for the computation of the maximum degree of subdeterminants of a specified order.) Another method, known in the control literature [15], [38], is to transform a given matrix to a row- or column-proper matrix. Comparison of theoretical and practical efficiency of various algorithms is left for future investigation.

Acknowledgments. The author thanks Masaaki Sugihara for discussion and Koichi Kubota and Akihiro Sugimoto for careful reading of the manuscript. The comments of the anonymous referees were helpful in revision.

REFERENCES

[1] R. K. AHUJA, T. L. MAGNANTI, AND J. B. ORLIN, *Network Flows—Theory, Algorithms, and Applications*, Prentice–Hall, Englewood Cliffs, NJ, 1993.

[2] M. O. BALL AND U. DERIG, *An analysis of alternative strategies for implementing matching algorithms*, Networks, 13 (1983), pp. 517–549.

[3] M. BEN-OR AND P. TIWARI, *A deterministic algorithm for sparse multivariate polynomial interpolation*, Proc. 20th Annual ACM Symposium on Theory of Computing, Chicago, 1988, pp. 301–309.

[4] A. BOUCHET, *Greedy algorithm and symmetric matroids*, Math. Programming, 38 (1987), pp. 147–159.

[5] R. A. BRUALDI AND H. J. RYSER, *Combinatorial Matrix Theory*, Cambridge University Press, London, 1991.

[6] B. BUCHBERGER, G. E. COLLINS, AND R. LOOS, *Computer Algebra—Symbolic and Algebraic Computation*, Computing Supplementum 4, Springer-Verlag, Berlin, 1982.

[7] R. CHANDRASEKARAN AND S. N. KABADI, *Pseudomatroids*, Discrete Math., 71 (1988), pp. 205–217.

[8] W. H. CUNNINGHAM AND A. B. MARSH III, *A primal algorithm for optimal matching*, Math. Programming Study, 8 (1978), pp. 50–72.

[9] J. DAVENPORT, Y. SIRET, AND E. TOURNIER, *Computer Algebra—Systems and Algorithms for Algebraic Manipulation*, 2nd ed., Academic Press, New York, 1993.

[10] A. DRESS AND T. HAVEL, *Some combinatorial properties of discriminants in metric vector spaces*, Adv. Math., 62 (1986), pp. 285–312.

[11] J. EDMONDS, *Paths, trees, and flowers*, Canad. J. Math., 17 (1965), pp. 449–467.

[12] ———, *Maximum matching and a polyhedron with* 0, 1*-vertices*, J. Res. Nat. Bur. Stand., 69B (1965), pp. 125–130.

[13] ———, *Systems of distinct representatives and linear algebra*, J. Res. Nat. Bur. Stand., 71B (1967), pp. 241–245.

[14] J. EDMONDS AND E. L. JOHNSON, *Matching: A well-solved class of integer programs*, in Combinatorial Structures and Their Applications, R. Guy et al., eds., Gordon and Breach, New York, 1970, pp. 89–92.

[15] G. D. FORNEY, JR., *Minimal bases of rational vector spaces with applications to multivariable linear systems*, SIAM J. Control, 13 (1975), pp. 493–520.

[16] H. N. GABOW, *Data structures for weighted matching and nearest common ancestors with linking*, Proc. 1st Annual ACM–SIAM Symposium on Discrete Algorithms, San Francisco, 1990, pp. 434–443.

[17] H. N. GABOW AND R. E. TARJAN, *Faster scaling algorithms for general graph-matching problems*, J. Assoc. Comput. Mach., 38 (1991), pp. 815–853.

[18] M. IRI, *Applications of matroid theory*, in Mathematical Programming—State of the Art, A. Bachem, M. Grötschel, and B. Korte, eds., Springer-Verlag, Berlin, 1983, pp. 158–201.

[19] M. IRI, S. FUJISHIGE, AND T. OHYAMA, *Graph, Network and Matroid*, Sangyo-tosho, Tokyo, 1986. (In Japanese.)

[20] Y. HAYAKAWA, S. HOSOE, AND M. ITO, *Dynamical degree and controllability for linear systems with intermediate standard form*, Trans. Inst. Electr. Comm. Engrg. Japan, J64A (1981), pp. 752–759. (In Japanese.)

[21] E. L. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart, and Winston, New York, 1976.

[22] L. LOVÁSZ AND M. PLUMMER, *Matching Theory*, North–Holland, Amsterdam, 1986.

[23] K. MUROTA, *Systems Analysis by Graphs and Matroids—Structural Solvability and Controllability*, in Algorithms and Combinatorics 3, Springer-Verlag, Berlin, 1987.

[24] ———, *Some recent results in combinatorial approaches to dynamical systems*, Linear Algebra Appl., 122–124 (1989), pp. 725–759.

[25] ———, *Structure-oriented algorithm for determining dynamical degree*, Tech. report 89591-OR, Forschungsinstitut für Diskrete Mathematik, Universität Bonn, Bonn, Germany, 1989.

[26] ———, *Computing Puiseux-series solutions to determinantal equations via combinatorial relaxation*, SIAM J. Comput., 19 (1990), pp. 1132–1161.

[27] ———, *An identity for matching and skew-symmetric determinant*, Linear Algebra Appl., 218 (1995), pp. 1–27.

[28] ———, *An identity for bipartite matching and symmetric determinant*, Linear Algebra Appl., to appear.

[29] K. MUROTA, *Combinatorial relaxation algorithm for the maximum degree of subdeterminants: Computing Smith–McMillan form at infinity and structural indices in Kronecker form*, Appl. Algebra Engrg. Comm. Comput., to appear.
[30] M. NEWMAN, *Integral Matrices*, Academic Press, New York, 1972.
[31] C. H. PAPADIMITRIOU AND K. STEIGLITZ, *Combinatorial Optimization: Algorithms and Complexity*, Prentice–Hall, Englewood Cliffs, NJ, 1982.
[32] A. RECSKI, *Matroid Theory and Its Applications in Electric Network Theory and in Statics*, in Algorithms and Combinatorics 6, Springer-Verlag, Berlin, 1989.
[33] H. H. ROSENBROCK, *State-Space and Multivariable Theory*, Nelson, London, 1970.
[34] T. SASAKI, *Symbolic and Algebraic Manipulation*, Inform. Process. Soc. Japan, Tokyo, 1981. (In Japanese.)
[35] A. SCHRIJVER, *Theory of Linear and Integer Programming*, John Wiley, Chichester, 1986.
[36] R. E. TARJAN, *Data Structures and Network Algorithms*, in SIAM Regional Conference Series in Applied Math. 44, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.
[37] W. T. TUTTE, *The factorization of linear graphs*, J. London Math. Soc., 22 (1947), pp. 107–111.
[38] W. A. WOLOVICH, *Linear Multivariable Systems*, Springer-Verlag, Berlin, 1974.

SIAM J. COMPUT.
Vol. 24, No. 4, pp. 797–810, August 1995

007

SCHEDULING TASKS WITH AND/OR PRECEDENCE CONSTRAINTS*

DONALD W. GILLIES† AND JANE W.-S. LIU‡

Abstract. In traditional precedence-constrained scheduling a task is ready to execute when all its predecessors are complete. We call such a task an AND task. In this paper we allow certain tasks to be ready when just one of their predecessors is complete. These tasks are known as OR tasks. We analyze the complexity of two types of real-time AND/OR task scheduling problems. In the first type of problem, all the predecessors of every OR task must eventually be completed, but in the second type of problem, some OR predecessors may be left unscheduled. We show that most problems involving tasks with individual deadlines are NP-complete, and then present two priority-driven heuristic algorithms to minimize completion time on a multiprocessor. These algorithms provide the same level of worst-case performance as some previous priority-driven algorithms for scheduling AND-only task systems.

Key words. nonpreemptive scheduling, list scheduling, minimal length schedules, algorithm analysis, multiprocessor systems, NP-complete problems, imprecise computation

AMS subject classifications. 68M20, 68Q25, 90B35, 90C90

1. Introduction.

In the traditional model of real-time workloads, dependencies between tasks are represented by partial orders known as precedence constraints. Each task may have several predecessors and may not begin execution until all its predecessors are completed. We call such tasks AND tasks, and the partial order over them is known as AND-only precedence constraints. This traditional model fails to describe many real-time applications encountered in practice. In these applications a task may begin execution when some but not all of its predecessors are completed. We call such a task an OR task. The resulting task system, containing both AND and OR tasks, is said to have AND/OR precedence constraints.

In this paper we are concerned with how to schedule tasks with AND/OR precedence constraints to meet deadlines. We investigate two variants of this problem, called the unskipped and the skipped variants.

In some applications all the predecessors of an OR task must eventually be completed, that is, they cannot be skipped. We call the model for this type of application the *AND/OR/unskipped* model. For example, in robotic assembly [1] one out of four bolts may secure an engine head well enough to allow further work on other parts of the engine head. However, the remaining three bolts must eventually be installed. The unskipped variant also models tasks that share resources. A task may need a resource from one of several predecessors in order to execute and hence is ready to execute when any one predecessor is complete. Such a task can be modeled as an OR task. Again, the other predecessors must eventually be completed. The AND/OR/unskipped problem also arises in hard real-time scheduling when the precedence constraints are too strict for tasks to meet their deadlines. By relaxing the precedence constraints of some tasks and restructuring the application code to accommodate the relaxed constraints, it may be possible for the tasks to meet their deadlines.

In other applications some predecessors of an OR task may be skipped entirely. We call this the *AND/OR/skipped* model. One example can be found in the problem of instruction scheduling on superscalar, multiple instruction multiple data (MIMD), or very long instruction word (VLIW) processors. On such processors several different instruction sequences may be used to compute the same arithmetic expression. These different sequences arise from algebraic laws such as associativity and distributivity. Only one sequence needs to be executed and the other sequences may be skipped. Another application that can be characterized by this

*Received by the editors August 26, 1991; accepted for publication (in revised form) February 28, 1994. This research was supported by Office of Naval Research contracts NVY N00014-89-J-1181 and NVY N00014-92-J-1146.

†Department of Electrical Engineering, University of British Columbia, Vancouver, British Columbia, Canada.

‡Department of Computer Science, University of Illinois, Urbana, Illinois 61801.

model is manufacturing planning [5] because certain manufacturing steps obey associative and distributive algebraic laws. The AND/OR/skipped problem also arises in hard real-time scheduling. When there is insufficient time for a task system to meet its deadlines, we may convert appropriate tasks to imprecise computations [3], which may be modeled as OR tasks whose predecessors may be skipped.

We are concerned with ways to schedule AND/OR precedence-constrained tasks to meet deadlines or minimize completion time. Most of these problems are generalizations of traditional deterministic scheduling problems that are NP-hard. In this paper we analyze the complexity of the problems that are not known to be NP-hard. For two problems that are known to be NP-hard, we give heuristic algorithms to minimize completion time. The algorithms have small running time and good worst-case performance.

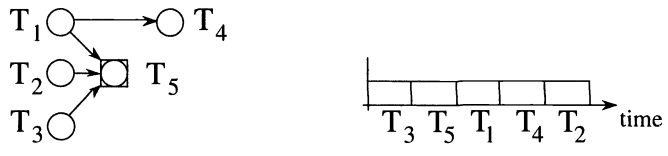
Our work is related to some previous work on deterministic scheduling to meet deadlines [6], [8] and minimize completion time [9], [10], [13], [14]. We were inspired by an AND/OR model that was proposed as a means of modeling distributed systems for real-time control [18]. Two recent systems incorporated AND/OR precedence constraints of some sort in their implementation [16], [19].

The remainder of this paper is organized as follows: Section 2 describes our assumptions about the AND/OR scheduling problem and introduces the terminology used in later sections. Section 3 investigates the unskipped problem with multiple deadlines and analyzes an algorithm to minimize completion time. In §4 we investigate the skipped problem and give a second algorithm to minimize completion time. Section 5 draws conclusions and discusses future work. The appendix contains proofs of the theorems stated in §§3 and 4.

2. The AND/OR model. All the scheduling problems considered here are variants of the following problem: There are m identical processors and a set of tasks $\mathbf{T} = \{T_1, T_2, \ldots, T_n\}$. Each task T_i must execute on one processor for p_i units of time and is said to have *processing time* p_i. There is a partial order $<$ defined over $\mathbf{T}$. If $T_i < T_j$, then T_i is a *predecessor* of T_j and T_j is a *successor* of T_i. The task T_i is a *direct predecessor* of T_j if there is no T_k such that $T_i < T_k < T_j$. The task T_j is an AND task if its execution may begin only after all its direct predecessors have completed their executions. The task T_j is an OR task if its execution may begin after only one of its direct predecessors has completed its execution. The partial order $<$ is an *in-forest* if, whenever $T_k < T_i$ and $T_k < T_j$, we have either $T_i < T_j$ or $T_j < T_i$; the partial order $<$ is an *in-tree* if it has a unique element with no successors. A task followed by a series of direct successors $T_{i_1} < T_{i_2} < \cdots$ is called a *task chain.*

The partial order is also represented by a vertex-weighted and transitively reduced directed graph $\mathbf{G} = (\mathbf{T}, \mathbf{A}, \mathbf{P})$ called the *task graph.* In this graph there is a vertex T_i for every task in the set $\mathbf{T}$. The set $\mathbf{A}$ is known as the *set of arcs.* If T_i is a direct predecessor of T_j in the partial order then $(T_i, T_j) \in \mathbf{A}$. The set $\mathbf{P} = \{p_1, \ldots, p_n\}$ denotes the set of processing times. A task graph together with a set of deadlines $\mathbf{D} = \{d_1, \ldots, d_n\}$ is a 2-tuple $(\mathbf{G}, \mathbf{D})$ that characterizes a scheduling problem; it is called a *task system.* When several graphs $\mathbf{G_1}, \mathbf{G_2}, \ldots$ are present, the functions $T(\mathbf{G_i})$, $A(\mathbf{G_i})$, and $P(\mathbf{G_i})$ will be used to extract the sets $\mathbf{T}$, $\mathbf{A}$, and $\mathbf{P}$ from the graph $\mathbf{G_i}$.

Let $S(\mathbf{G}, T_i) = \{T_j | (T_i, T_j) \in A(\mathbf{G}), T_i \in T(\mathbf{G})\}$ denote the set of direct successors of T_i, and let $P(\mathbf{G}, T_i) = \{T_j | (T_j, T_i) \in A(\mathbf{G}), T_i \in T(\mathbf{G})\}$ denote the set of direct predecessors of T_i. Let $L(\mathbf{G}, T_j)$ be the length of the longest directed path in $\mathbf{G}$ ending at T_j. More precisely, $L(\mathbf{G}, T_j) = p_j$ if T_j has no predecessors in $\mathbf{G}$, and $L(\mathbf{G}, T_j) = p_j + \max_k\{L(\mathbf{G}, T_k) | (T_k, T_j) \in A(\mathbf{G})\}$ if T_j has predecessors. Let $L^*(\mathbf{G}) = \max\{L(\mathbf{G}, T_j) | T_j \in T(\mathbf{G})\}$ be the length of the longest directed path in a graph $\mathbf{G}$. Let $E(\mathbf{G}, T_i) = \sum_{T_j < T_i \text{ in } \mathbf{G}} p_j$ denote the total processing time of all the predecessors of T_i in $\mathbf{G}$. Let $E^*(\mathbf{G}) = \sum p_i - L^*(\mathbf{G})$ denote the "residual" processing time of an AND-only graph, i.e., the total processing time minus the processing



(a) *Task system.* (b) *AND/OR/unskipped schedule.*

FIG. 1. *Sample problem and solution.*

time of the tasks on the longest chain. Later it will be shown that AND-only graphs with minimal $L^*(\mathbf{G})$ and $E^*(\mathbf{G})$ can be used to produce near-optimal priority-driven schedules.

All the tasks with no successors in a task graph are classified as *essential*; this means that they must appear in a valid schedule. If an AND task is essential then all its direct predecessors are essential. If an OR task T_j is essential then the scheduling algorithm must choose one direct predecessor T_i to be essential and the precedence constraint $T_i < T_j$ must be obeyed in scheduling the task system. If a task is not classified as essential by any successor then it is *inessential*. We distinguish between two problems referred to as skipped and unskipped problems, respectively. In a skipped scheduling problem inessential tasks may be left unexecuted. However, in an unskipped problem inessential tasks must be executed.

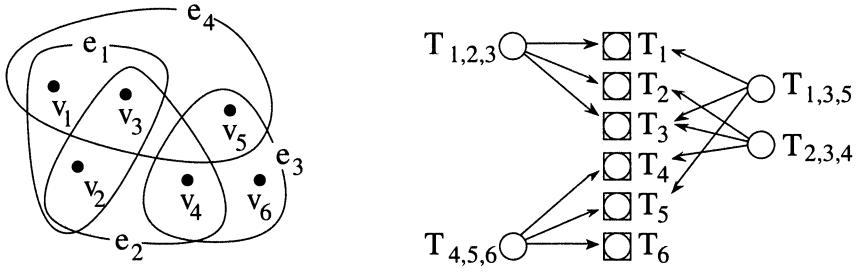
Figure 1(a) depicts an AND/OR task system. In the figure AND tasks are depicted by circles and OR tasks are depicted by circles within boxes. Tasks are generally labeled by their *name* or by their (*name length*), so (T_5, e) would indicate that task T_5 requires e units of processing time. Where necessary, deadlines will be written separately, next to the associated tasks. If the deadlines are omitted from a figure, the reader should assume that all the deadlines are identical. Every task in this example has a processing time of one and all the tasks have the same deadline, hence the lengths and deadlines are omitted from this figure. Figure 1(b) depicts a schedule in which T_3 is an essential task and T_2 is an inessential task. Figure 1(b) shows a schedule of the unskipped task graph from Figure 1(a). If Figure 1(a) depicted a skipped task graph then a skipped schedule could be obtained by deleting T_2 from the end of the schedule in Figure 1(b).

The scheduling algorithms in this paper are simple heuristics that never intentionally leave processors idle. These algorithms are known as *priority-driven* or *list-scheduling* algorithms. Whenever a processor is available, a list-scheduling algorithm schedules the ready task with the highest priority according to a priority list. Because they try to make the best local choice at each scheduling decision point, list-scheduling algorithms are also called greedy algorithms. A schedule produced by a list-scheduling algorithm is known as a *list schedule*, and the time at which all the tasks in $\mathbf{T}$ are complete is the *length* of the schedule.

We assume that every task in $\mathbf{T}$ has ready time equal to zero, thus an OR task may begin execution as soon as an essential predecessor is completed. In some situations each task T_i has a deadline d_i; T_i must be completed at or before time d_i. A schedule is called *feasible* if every task completes by its deadline. A task system that has a feasible schedule is called *feasible*. Given a task system, our objective is to find a feasible schedule or determine that no feasible schedule exists.

In other situations all the tasks share a common deadline. The problem of finding a feasible schedule in these situations is equivalent to the problem of minimizing the overall completion time, i.e., the time at which the last task completes.

3. Unskipped problems. In this section we discuss the complexity of the AND/OR/unskipped scheduling problem. After showing that most natural problems with deadlines are NP-complete on a single processor, we present a priority-driven heuristic to



(a) *Exact 3-cover problem.* (b) *AND/OR task system.*

FIG. 2. *Exact 3-cover transformation.*

minimize completion time on m processors. We then explain why no priority-driven heuristic can provide a better worst-case performance bound than the one presented here.

3.1. Scheduling to meet deadlines on a single processor. There are well-known polynomial-time algorithms [6], [8] for scheduling tasks with AND-only precedence constraints, identical processing times, and arbitrary deadlines on one or two processors. It is natural to ask whether the corresponding AND/OR scheduling problems may be solved in polynomial time. Unfortunately, this extended problem is NP-complete, even when there are only two deadlines. This fact is expressed in the following theorem.

THEOREM 3.1. *The problem of AND/OR skipped or unskipped scheduling of a task system in which all the OR tasks must meet a common deadline is NP-complete.*

Proof. It suffices to prove that the problem is NP-complete on a single processor. The proof is based on a reduction from exact 3-cover (X3C). Given a hypergraph $H = (V, E)$ of $3n$ vertices and a set of hyperedges, each of which is incident to three vertices, the problem is to find a set of exactly n edges that covers all the vertices with no overlap. This problem is NP-complete [7].

The exact 3-cover problem can be transformed into an AND/OR scheduling problem as follows: Create a task system $(\mathbf{G}, \mathbf{D})$ composed entirely of unit processing-time tasks. There is an OR task T_i in the task system for each hypergraph vertex v_i in H. In the task system all $3n$ OR tasks have deadline $4n$. Create an AND task $T_{i,j,k}$ for each hyperedge that connects v_i, v_j, and v_k. The successors of task $T_{i,j,k}$ are the OR tasks T_i, T_j, and T_k. Figure 2 is an example of this transformation. Now we ask if there exists a schedule in which every OR task meets its deadline. Clearly, if the given hypergraph H has an exact 3-cover, n AND tasks corresponding to the cover may execute in the time interval $[0, n]$, thereby allowing all $3n$ OR tasks to complete by time $4n$. If no such cover exists, then at least $n + 1$ edges must be used to cover the hypergraph. Hence at least $n + 1 + 3n$ time units must elapse before all the OR tasks are complete regardless of whether this a skipped or an unskipped problem. Thus, if a scheduler produces a feasible schedule then there is an exact 3-cover, and if the scheduler fails then no such cover exists. □

The proof of Theorem 3.1 indicates that this scheduling problem is at least as hard as the n-dimensional cover problem, a generalized version of n-dimensional matching. About thirty years ago, T. C. Hu gave a polynomial-time algorithm to schedule an AND-only task system with in-tree precedence constraints on m processors [14]. Thus, there is some hope that if we restrict the AND/OR/unskipped task system to have in-tree precedence constraints, there may exist a polynomial-time algorithm. Unfortunately, the following theorem shows that this AND/OR scheduling problem is NP-complete.

TABLE 1
Complexity of AND/OR/unskipped problems.

(a) *Scheduling to meet deadlines with identical processing times on 1 processor.*

Deadline location	General graph 2 deadlines	In-tree $O(n)$ deadlines	Simple in-forest
on all tasks	NP-C (Theorem 3.1)	NP-C (Theorem 3.2)	NP-C (Theorem 3.3)
on OR tasks only	NP-C (Theorem 3.1)	NP-C (Corollary 3.1)	trivial

(b) *Scheduling to minimize completion time on m processors.*

Task processing time	General graph	In-tree
identical	NP-C [15] for AND-only	NP-C (Theorem 3.4)
arbitrary	minimum-path heuristic	minimum-path heuristic

THEOREM 3.2. *The problem of AND/OR/unskipped scheduling to meet deadlines, where tasks have identical processing times, arbitrary deadlines, and in-tree precedence constraints, is NP-complete.*

Proof. The proof is contained in the appendix. □

COROLLARY 3.1. *The problem remains NP-complete for task systems in which only the OR tasks have deadlines.*

Proof. The proof is contained in the appendix. □

The proofs of Theorem 3.2 and Corollary 3.1 in the appendix make use of long chains of AND tasks with differing deadlines. We now consider a class of task systems where only two tasks in a chain may have deadlines. In a *simple in-forest*, (1) each in-tree consists of an OR task with a deadline, no successors, and two direct predecessors, and (2) each direct predecessor of an OR task has a deadline and is the root of an in-tree of AND tasks with no deadlines (i.e., the deadlines are infinite). A simple in-forest restricts the allowable precedence constraints and allowable tasks with deadlines in a task system. We have found no simpler nontrivial combination of precedence constraints and deadlines. Surprisingly, even this simplified AND/OR scheduling problem is NP-complete

THEOREM 3.3. *The problem of AND/OR/unskipped scheduling to meet deadlines, where the task system is a simple in-forest with identical processing times, is NP-complete.*

Proof. The proof may be found in [11] or [12]. □

Theorems 3.1–3.3 allow us to arrive at the following conclusion: Every AND/OR task graph with k OR tasks, each of which has l direct predecessors, corresponds to a set of l^k different AND-only task graphs. A feasible schedule of such a task system corresponds to an implicit selection of one of these l^k AND-only task graphs. Therefore, when there are $O(\log n)$ OR tasks in the AND/OR task system, it is possible to enumerate in polynomial time the set of all possible AND-only task graphs and apply an optimal AND-only scheduling algorithm such as the one described in [8]. On the other hand, Theorems 3.1–3.3 show that many natural scheduling problems with $O(n)$ OR tasks are NP-complete. It follows that the complexity of the AND/OR/unskipped problem is determined almost exclusively by the number of OR tasks in the task system and the complexity of the corresponding AND-only scheduling problem. These results are summarized in Table 1(a).

It appears difficult to design a priority-driven scheduling heuristic with good worst-case performance. For the simple problem studied in Theorem 3.3, we have produced examples to show that any algorithm that only considers slacks between deadlines and nondeadline information, one isolated in-tree at a time, may perform $\sqrt{n}$ times worse than an optimal

Input: Task graph $\mathbf{G} = (\mathbf{T}, \mathbf{A}, \mathbf{P})$
Step 1: For each OR task T_i with no OR predecessors:
(a) Let T_k be a direct predecessor of T_i that minimizes the longest path ending at T_k. In other words, $T_k \in P(\mathbf{G}, T_i)$ and for all $T_j \in P(\mathbf{G}, T_i)$ with $j \neq k$, $L(\mathbf{G}, T_j) \geq L(\mathbf{G}, T_k)$.
(b) Convert T_i into an AND task whose only direct predecessor is T_k .
Step 2: The resulting task system has only AND tasks. Schedule this task system using a priority-driven algorithm and an arbitrary priority list.

FIG. 3. *The minimum path heuristic for general graphs.*

algorithm. Some obvious priority-driven scheduling algorithms such as fewest predecessors first, least slack first, and some generalizations of the algorithms in [4] neglect to compare the deadlines among different in-trees. In the worst case these algorithms may meet only $\sqrt{n}$ deadlines when it is possible to meet n out of $n+1$ deadlines. For more information the reader is referred to [11] and [12].

3.2. Scheduling to minimize completion time. We now consider the problem of scheduling AND/OR/unskipped tasks with arbitrary processing times on m processors to meet a common deadline. This problem is equivalent to that of scheduling to minimize the overall completion time. Ullman has shown this problem to be NP-complete [15] for AND-only task systems where all the tasks have identical processing times. However, Hu's algorithm solves this problem in polynomial time for in-tree precedence constraints. Unfortunately, the problem becomes NP-complete when OR tasks are allowed.

THEOREM 3.4. *The problem of scheduling an AND/OR/unskipped task system to minimize completion time on m processors, where tasks have identical processing times and in-tree precedence constraints, is NP-complete.*

Proof. The proof is contained in the appendix. □

In Figure 3, we present a heuristic that minimizes the completion time of an AND/OR/unskipped task system with arbitrary processing times. The basic idea is to choose an AND-only graph that minimizes the longest path in $\mathbf{G}$. The heuristic can be implemented to run in time $O(n+|\mathbf{A}|)$ by reversing the direction of the arcs in $\mathbf{G}$ and employing depth-first search. Let $\mathbf{G_o} = (\mathbf{T_o}, \mathbf{A_o}, \mathbf{P_o})$ and W_o denote the implicit AND-only graph and the completion time of the task system according to an optimal schedule, respectively. Let $\mathbf{G}' = (\mathbf{T}', \mathbf{A}', \mathbf{P}')$ and W' denote the implicit AND-only graph and the completion time of the task system according to a schedule produced by the minimum path heuristic, respectively. The worst-case performance of the minimum path heuristic depends on the following lemma.

LEMMA 3.1. $L^*(\mathbf{G}') \leq L^*(\mathbf{G_o})$.

Proof. Let $H = \{T_i | P(\mathbf{G}', T_i) \neq P(\mathbf{G_o}, T_i)\}$ denote the set of tasks whose predecessors differ between the optimal graph and the graph produced in Step 1 of the minimum path heuristic. If $H = \emptyset$, then the AND-only task graphs are identical and the lemma is established. Otherwise, let $T_i \in H$ be a task for which there exists no $T_j \in H$ with $T_j < T_i$ in $\mathbf{G_o}$. By the construction of $\mathbf{G}'$, $|P(\mathbf{G}', T_i)| = |P(\mathbf{G_o}, T_i)| = 1$. We change $\mathbf{A_o}$, replacing the arc $(P(\mathbf{G_o}, T_i), T_i)$ by $(P(\mathbf{G}', T_i), T_i)$, and obtain no increase in the longest path (by Step 1(a) and (b) of the heuristic). This argument is used inductively to transform $\mathbf{G_o}$ into $\mathbf{G}'$ with no increase in the maximum path length. This establishes the lemma. □

The following fact is proved in the well-known paper [13].

LEMMA 3.2. *In any priority-driven schedule, there is a chain of tasks that executes during all the idle periods (when one or more processors are not in use), and this chain is not longer than the completion time of an optimal schedule.*

If W_p denotes the total length of all the idle periods in a schedule produced by the minimum path heuristic, then $W_p \leq L^*(\mathbf{G}') \leq L^*(\mathbf{G_0}) \leq W_o$ by Lemmas 3.1 and 3.2.

THEOREM 3.5. *The worst-case performance of the minimum path heuristic is given by* $W'/W_o \leq 2 - 1/m$. *Moreover, this bound is tight.*

Proof. Let W_b denote the total length of all the busy periods in a priority-driven schedule. Let W_p denote the total length of all the idle periods in a priority-driven schedule. During the idle periods, at least 1 and no more than $m - 1$ tasks execute, and during the busy periods exactly m tasks execute. It should be clear that $W' = W_p + W_b$. Hence, the worst-case completion time of this heuristic may be formulated as a linear program:

$$\begin{aligned} &\text{Maximize} && W_p + W_b = W' \\ &\text{subject to} && W_p \leq L^*(\mathbf{G}') \leq L^*(\mathbf{G_0}) \leq W_o \\ & && mW_b + 1W_p \leq mW_o. \end{aligned}$$

Solving the program yields $W_p = W_o$, $W_b = (1 - 1/m)W_o$, i.e., $W'/W_o \leq 2 - 1/m$. □

Examples of AND-only task systems that achieve this bound may be found in [2] and [10]. It is known [10] that no AND-only priority-driven heuristic can avoid $2 - 1/m$ worst-case performance (because priority-driven heuristics never intentionally idle the processor, and sometimes intentional idling is needed). Our priority-driven heuristic will schedule AND-only task systems as a special case. Hence, it is not possible to get better worst-case performance from an AND/OR scheduling algorithm without a better AND-only scheduling algorithm. It has been a long-standing open problem to find a better AND-only scheduling algorithm [15].

4. Skipped problems. In an AND/OR/skipped scheduling problem, the inessential predecessors of an OR task may be skipped entirely. We first show that when the problems of §3 are formulated in the skipped model they remain NP-complete. Then we present a heuristic algorithm for scheduling to minimize completion time on m processors. This heuristic algorithm works for in-tree precedence constraints, but not for arbitrary precedence constraints.

4.1. Scheduling to meet deadlines. Theorem 3.1 showed that the problem of AND/OR/skipped scheduling with one deadline and arbitrary precedence constraints is NP-complete on a single processor, therefore, we immediately consider simplifying the precedence constraints.

THEOREM 4.1. *The problem of AND/OR/skipped scheduling to meet deadlines, where tasks have identical processing times and in-tree precedence constraints, is NP-complete.*

Proof. The proof is contained in the appendix. □

THEOREM 4.2. *The problem of AND/OR/skipped scheduling to meet deadlines, where the task system is a simple in-forest with identical processing times, is NP-complete.*

Proof. The proof may be found in [11] or [12]. □

Now we consider the case where the task system is a simple in-forest and only the OR tasks have deadlines. For this type of task system, an algorithm for finding a feasible schedule can examine each OR task and choose as its direct predecessor the AND task which has the fewest total predecessors. After these choices are made, the AND-only graph is scheduled using the earliest deadline first rule. This method always produces a feasible schedule if the task system is feasible. If the task system is infeasible it is still possible to maximize the number of OR tasks that simultaneously meet their deadlines and have essential predecessors. To produce such a schedule we note that an OR task, together with one predecessor subtree consisting of k_i AND tasks, may be thought of as one large task with processing time $k_i + 1$. Then the algorithm of [17], which minimizes unit penalty on a single processor, may be used

TABLE 2
Complexity of AND/OR/skipped problems.

(a) *Scheduling to meet deadlines with identical processing times on* 1 *processor.*

Deadline location	General graph 1 deadline	In-tree O(n) deadlines	Simple in-forest
on all tasks	NP-C (Theorem 3.1)	NP-C (Theorem 4.1)	NP-C (Theorem 4.2)
on OR tasks only	NP-C (Theorem 3.1)	NP-C (Theorem 4.1)	[17] algorithm

(b) *Scheduling to minimize completion time on m processors.*

Task processing time	General graph	In-tree
identical	NP-C [15] ($\geq 3/2 * OPT$)	NP-C (Theorem 4.3)
arbitrary	no algorithm	path-balancing heuristic

to schedule tasks with processing time $(k_i + 1)$ to maximize the number of OR tasks that meet their deadline.

In summary, we find that the complexity of the skipped problem is always at least as high as the complexity of the unskipped problem. This fact is summarized in Table 2(a).

4.2. Scheduling to minimize completion time. Table 2(b) gives the complexity of scheduling m processors to minimize completion time. The next theorem concludes our investigation into the complexity of AND/OR scheduling.

THEOREM 4.3. *The problem of scheduling an AND/OR/skipped task system to minimize completion time on m processors, where tasks have identical processing times and in-tree precedence constraints, is NP-complete.*

Proof. The proof is contained in the appendix. □

Now we present a heuristic algorithm that minimizes the completion time of an AND/OR/skipped task system with in-tree precedence constraints. Let $f(\mathbf{G}) = E^*(\mathbf{G})/m + L^*(\mathbf{G})$ denote a function of an AND-only precedence graph. This function is an estimate of the worst-case completion time of a priority-driven schedule. Our algorithm converts an AND/OR in-tree into an AND-only in-tree that minimizes this function. In a general graph it is difficult to minimize this function quickly. If $m = 1$, a polynomial-time algorithm to minimize $f(\mathbf{G})$ could be used to solve any exact 3-cover problem (refer to Theorem 3.1), implying P = NP. Because of this, the path-balancing heuristic described below is restricted to in-tree task graphs. The algorithm appears in Fig. 4.

The complexity of the algorithm can be determined as follows. The $O(n)$ possible paths from the root to the leaves can be enumerated in time $O(n)$ using depth-first search. Each iteration of Step 1(a)–(e) can be carried out together in $O(n)$ time using a recursive depth-first search. Most of the work is done when returning from procedure calls. Hence, the overall complexity of this heuristic is $O(n^2)$.

To derive the worst-case performance of the path-balancing heuristic we begin by showing that Step 1 of this heuristic minimizes $f()$.

LEMMA 4.1. $f(\mathbf{G}') \leq f(\mathbf{G_0})$.

Proof. Consider the longest path of length $L^*(\mathbf{G_0})$ in $\mathbf{G_0}$. This path starts at the tree root and ends at a leaf vertex. Clearly, the path-balancing heuristic considers this path in some iteration of Step 1. Step 1(c) of the heuristic ensures that no other paths are longer than this longest path without increasing $E^*(\mathbf{G}')$ more than necessary. Step 1(d) of the heuristic chooses the direct predecessors of each OR task to minimize $E^*(\mathbf{G}')$, thus the heuristic cannot fail to find a graph for which $f(\mathbf{G}')$ is at most $E^*(\mathbf{G_0})/m + L^*(\mathbf{G_0})$. □

Input: Task graph $\mathbf{G} = (\mathbf{T}, \mathbf{A}, \mathbf{P})$

Step 1: Convert the OR tasks in the in-tree $\mathbf{G}$ into AND tasks, to obtain an AND-only graph $\mathbf{G}'$ that minimizes $f(\mathbf{G}')$, as follows.

For each path $C_i = \{T_{x_1} < T_{x_2} < \cdots < T_{x_k}\}$ from the root to a leaf in $\mathbf{G}$ do begin

(a) [Copy $\mathbf{G}$] $\mathbf{G_c} \leftarrow \mathbf{G}$.

(b) [Freeze OR tasks along path C_i] For each OR task $T_{x_j} \in C_i$ let $\mathbf{A_c} = (\mathbf{A_c} - P(\mathbf{G_c}, T_{x_j})) \cup \{(T_{x_{j-1}}, T_{x_j})\}$ (i.e. make T_{x_j} an AND task in $\mathbf{G_c}$).

(c) [Truncate all paths longer than C_i] Let $C_j \neq C_i$ be a longer path in $\mathbf{G_c}$. If no such C_j exists, go to Step (d). Otherwise, let T_k be the least OR task on C_j. If no such T_k exists then go to Step (f). For each $T_l \in P(\mathbf{G_c}, T_k)$ on a path longer than C_i, do begin remove the arc (T_l, T_k) from $\mathbf{G_c}$ end. If $|P(\mathbf{G_c}, T_k)| = 0$ no AND-only graph exists with C_i as the longest path, so go to Step (f). Else Repeat Step (c).

(d) [Minimize processing time] For each OR task T_k with 2 or more direct predecessors and no OR predecessors in the graph $\mathbf{G_c}$, pick as a sole predecessor of T_k the task $T_j \in P(\mathbf{G_c}, T_k)$ such that for all $T_i \in P(\mathbf{G_c}, T_k)$ with $i \neq j$, $E(\mathbf{G_c}, T_i) \geq E(\mathbf{G_c}, T_j)$.

(e) If the resulting AND-only graph yields a value $f(\mathbf{G_c}) < f(\mathbf{G}')$ then let $\mathbf{G}' \leftarrow \mathbf{G_c}$.

(f) end.

Step 2: The resulting task system $\mathbf{G}'$ contains only AND tasks. Schedule this task system using a priority-driven heuristic and an arbitrary priority list.

FIG. 4. *The path-balancing heuristic for in-trees.*

THEOREM 4.4. *The worst-case performance of the path-balancing heuristic is given by*

$$\frac{W'}{W_o} \leq 2 - \frac{1}{m}. \tag{1}$$

Moreover, this bound is tight.

Proof. Any optimal schedule completes no earlier than the total processing time of the task system divided by m processors, and no earlier than $L^*(\mathbf{G_0})$. Hence

$$W_o \geq \max\left\{\frac{E^*(\mathbf{G_0}) + L^*(\mathbf{G_0})}{m}, L^*(\mathbf{G_0})\right\},$$

and by Lemmas 3.2 and 4.1, we have

$$W' \leq E^*(\mathbf{G}')/m + L^*(\mathbf{G}') \leq E^*(\mathbf{G_0})/m + L^*(\mathbf{G_0}).$$

Hence

$$\frac{W'}{W_o} \leq \frac{E^*(\mathbf{G_0})/m + L^*(\mathbf{G_0})}{\max\left\{\frac{E^*(\mathbf{G_0}) + L^*(\mathbf{G_0})}{m}, L^*(\mathbf{G_0})\right\}}. \tag{2}$$

We simplify equation (2) in two cases.

Case 1. The max{} in (2) evaluates to its first argument. Then we have

$$\frac{W'}{W} \leq \frac{E^*(\mathbf{G_0}) + L^*(\mathbf{G_0})m}{E^*(\mathbf{G_0}) + L^*(\mathbf{G_0})} = B. \tag{3}$$

Note that the max{} in (2) evaluates to its first argument if and only if $L^*(\mathbf{G_0}) \leq E^*(\mathbf{G_0})/(m - 1)$, so we have an upper bound on $L^*(\mathbf{G_0})$. The derivative of the bound in (3) is

$$\frac{dB}{dL^*(\mathbf{G_0})} = \frac{E^*(\mathbf{G_0})(m-1)}{2(E^*(\mathbf{G_0}) + L^*(\mathbf{G_0}))} \geq 0. \tag{4}$$

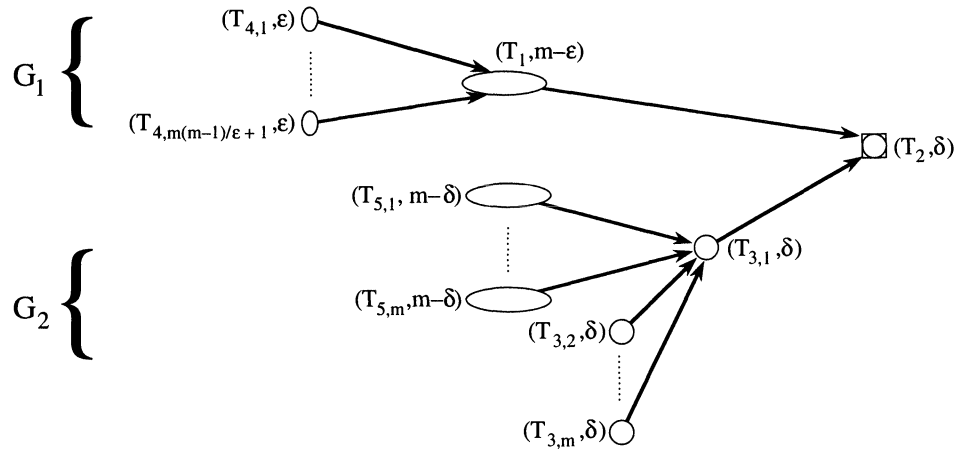


FIG. 5. *A worst-case AND/OR/skipped in-tree.*

Because the derivative of (4) is nonnegative for all $m \geq 1$ and $E^*(\mathbf{G_0}) \geq 0$, a maximum of (3) occurs when $L^*(\mathbf{G_0})$ is as great as possible, i.e., $L^*(\mathbf{G_0}) = E^*(\mathbf{G_0})/(m-1)$, thus

$$\frac{W'}{W} \leq \frac{E^*(\mathbf{G_0})(m-1) + E^*(\mathbf{G_0})m}{E^*(\mathbf{G_0})(m-1) + E^*(\mathbf{G_0})} = 2 - \frac{1}{m}.$$

Case 2. The max{} in (2) evaluates to its second argument. This occurs if and only if $E^*(\mathbf{G_0}) \leq L^*(\mathbf{G_0})(m-1)$. We substitute $E^*(\mathbf{G_0}) \leq L^*(\mathbf{G_0})(m-1)$ into the numerator of (2) to obtain (1). □

The example in Fig. 5 demonstrates that this worst-case bound is tight. Let $\mathbf{T_1} = \{T_1, T_2, T_{4,1}, \ldots, T_{4,m(m-1)/e+1}\}$ and let $\mathbf{T_2} = \{T_2, T_{3,1}, \ldots, T_{3,m}, T_{5,1}, \ldots, T_{5,m}\}$. The path-balancing heuristic chooses between the in-trees $\mathbf{G_1} = (\mathbf{T_1}, \mathbf{A_1}, \mathbf{P_1})$ and $\mathbf{G_2} = (\mathbf{T_2}, \mathbf{A_2}, \mathbf{P_2})$, where $\mathbf{A_1}$ and $\mathbf{A_2}$ denote the associated arc sets. The lengths of the longest paths in these in-trees are $L^*(\mathbf{G_1}) = L^*(\mathbf{G_2}) = m + \delta$, respectively. Furthermore, $E^*(\mathbf{G_1}) = E^*(\mathbf{G_2}) = m^2 - m$. Thus, the path-balancing heuristic chooses arbitrarily between these two trees, since either one minimizes $f(\mathbf{G}')$. There is a schedule of length $m + 2\delta$ for $\mathbf{G_2}$, but the shortest possible schedule for $\mathbf{G_1}$ has length $m + m(m-1)/m + \delta$ whenever ε divides $(m-1)$ evenly. As $d \to 0$, the ratio of these schedule lengths approaches $2 - 1/m$.

We now offer additional evidence that the problem of scheduling AND/OR/skipped task systems is much harder than the problem of scheduling AND-only task systems. Consider scheduling an AND/OR/skipped task system derived from an exact 3-cover problem, as described in the proof of Theorem 3.1, on a $(3n+1)$-processor system. We add to the task system an AND task with $2n + 1$ direct predecessors and ask if there is a schedule that completes in 2 units of time on $3n + 1$ processors. The task system is feasible if n tasks corresponding to edges in an exact 3-cover, together with the additional $2n + 1$ AND tasks, begin processing at time 0, and all the tasks corresponding to hypergraph vertices, together with the other added AND task, begin their processing at time 1. Hence there is a schedule with a completion time of 2 if and only if there is an exact 3-cover. It follows that unless P = NP, no polynomial-time AND/OR/skipped scheduling heuristic can guarantee a worst-case completion time of less than 3/2 times the length of an optimal schedule. In contrast to this, if the task system is AND-only, it is known [15] that no polynomial-time heuristic can guarantee a worst-case completion time of less than 4/3 times the length of an optimal schedule.

5. Conclusion. We have analyzed the skipped and unskipped variants of the AND/OR scheduling problem with deadlines. In the skipped variant, some tasks may be left unscheduled, but in the unskipped variant all tasks must be scheduled. When tasks had identical processing times, and deadlines, and there is a single processor, the problem was shown to be NP-complete, even for drastically simplified precedence constraints. We presented an efficient priority-driven heuristic to minimize completion time on m processors, and showed that its worst-case performance bound could not be improved by using a different priority-driven heuristic. We also presented a heuristic to minimize the completion time of an AND/OR/skipped task system with in-tree precedence constraints. We derived the worst-case performance for this algorithm and explained why the algorithm could not be extended to handle general task graphs with the same performance unless P = NP.

Throughout this paper we assumed that only one direct predecessor task had to be completed before an OR task could begin. Under a more general assumption, OR task T_i can begin once k_i predecessor tasks are complete. The algorithms and theorems in this paper require minor modifications to handle this more general case. There is also a similar AND/OR model where individual arcs (and not tasks) can be AND arcs or OR arcs. By using tasks with a processing time of zero, our model can simulate this other model. There are also situations where both OR/skipped and OR/unskipped tasks are present in a single in-tree. With slight modifications our AND/OR/skipped heuristic can be used to handle such mixed task systems. Details of these transformations and algorithms appear in [12].

During this investigation we reached several conclusions about the complexity of AND/OR scheduling. Contrary to our intuition, the skipped problems we considered were generally of higher complexity than the corresponding unskipped problems. This can be seen by comparing Tables 1 and 2, and the proofs in the appendix. In the problem of scheduling to meet deadlines, we made several observations. It was generally not helpful to restrict the in-degree of OR tasks in the task graph. It was also not helpful to restrict deadlines to only the OR tasks, or to restrict the task graph to an in-tree or an in-forest or even a simple in-tree, the simplest relation possible for this type of problem.

Appendix. This appendix presents the proofs of Theorems 3.2, 3.4, 4.1, 4.3, and Corollary 3.1. Proofs of Theorems 3.3 and 4.2 may be found in both [11] and [12]. Except where noted, all proofs refer to the scheduling of a single processor.

THEOREM 3.2. *The problem of AND/OR/unskipped scheduling to meet deadlines, where tasks have identical processing times, arbitrary deadlines, and in-tree precedence constraints, is NP-complete.*

Proof. Our proof is based on a reduction from 3-satisfiability (3SAT). Given an instance of a 3SAT problem with k boolean variables and n clauses, we will create k OR tasks. For each variable x_i, which occurs in l_i clauses, we create an in-tree containing one OR task and two chains of length l_i. One chain corresponds to truth for the associated variable and the other corresponds to falsity. Therefore, there are $3n$ tasks in all chains corresponding to truth and $3n$ tasks in all chains corresponding to falsity. The OR tasks are given deadlines of $e = 3n + k$. An example is shown in Fig. 6. This example is an in-tree for a variable x that appears in four clauses. Deadlines are depicted above or below the tasks. Because of the deadlines of the OR tasks, in any feasible schedule k OR tasks and k chains execute throughout the time interval $[0, e]$, and no other tasks may execute in this interval. This leaves k task chains to execute in the time period $[e, e + 3n]$ in a feasible schedule.

For each 3SAT clause we assign an interval of three time units starting at time e. Hence the time intervals $[e, e + 3], [e + 3, e + 6], \ldots, [e + 3n - 3, e + 3n]$ correspond to clause 1, clause 2, ..., clause n, respectively. Each interval of time is divided into two parts. In the first two time units, tasks in leftover chains corresponding to truth or falsity in a clause may

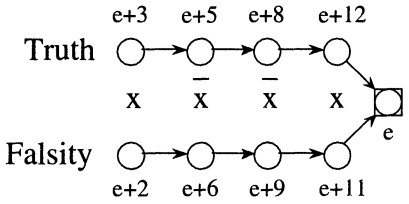


FIG. 6. *An in-tree for a variable x appearing in the first four clauses.*

execute. In the third time unit, only a task corresponding to truth may execute. To enforce this rule, we give later deadlines to the tasks/terms that would make each clause true. In Fig. 6, variable x occurs in the first four clauses of the 3SAT expression. It appears uncomplemented in clauses 1 and 4, and complemented in clauses 2 and 3. If x appears in the 3SAT expression for the ith time as an uncomplemented variable in clause j, the deadline for the ith task in the truth predecessor chain is $e+3j$; it is also $e+3j-1$ for the ith task in the falsity predecessor chain. These deadlines are exchanged if the ith appearance of x is as a complemented variable in clause j. We give all the OR tasks a common AND successor with a deadline of infinity to form a single tree.

If a scheduling algorithm finds a feasible schedule, then each task that executes in the interval $[e+3j-1, e+3j]$ corresponds to a variable (or a complemented variable) that is true in clause j. If the variable is not true, then the deadline of the task expires one time unit earlier. Furthermore, the task chains guarantee that the truth or falsity of a variable is consistent among different 3SAT clauses. Thus, a schedule is feasible if and only if there is a satisfying truth assignment. □

All the other proofs in this appendix and in [11] and [12] are modifications of the proof of Theorem 3.2. In particular, Theorems 3.3 and 4.2 require a large simple in-tree for each term in a 3SAT expression and have been omitted for brevity.

COROLLARY 3.1. *The problem remains NP-complete if only the OR tasks have deadlines.*

Proof. We make the following changes to the proof of Theorem 3.2: replace the in-trees of the type depicted by Fig. 6 by new in-trees such as the one in Fig. 7. This is done by adding an AND task with a deadline of e to the beginning of each truth and falsity chain, converting each AND task with a deadline into an OR task with one or two extra AND predecessor tasks, and setting $e = 3n + 5k$. As in the previous proof, the last deadline associated with a variable in a clause is $e + 3n$. Note that there are exactly $e + 3n$ tasks of the type that are shaded in Fig. 7 in the entire task system. Becuase of their deadlines, the shaded tasks must execute in the time interval $[0, e + 3n]$ and the unshaded tasks must execute after time $e + 3n$ in any feasible schedule.

It is not difficult to verify that in a feasible schedule the tasks that execute in the time interval $[e, e + 3n]$ correspond to a satisfying 3SAT truth assignment. □

THEOREM 4.1. *The problem of AND/OR/skipped scheduling to meet deadlines, where tasks have identical processing times, arbitrary deadlines on the OR tasks only, and in-tree precedence constraints, is NP-complete.*

Proof. This theorem extends the previous corollary to skipped tasks. We use nearly the same in-trees as in Corollary 3.1. However, we set $e = 2k$, give the OR tasks at the root of each in-tree a deadline of $e + 3n + k$ rather than e, and replace each unshaded task by a chain of e AND tasks. It is not difficult to check that the task chains that execute in the time intervals $[e + 3j - 1, e + 3j]$, $1 \leq j \leq n$, correspond to a truth assignment satisfying the 3SAT clauses. □

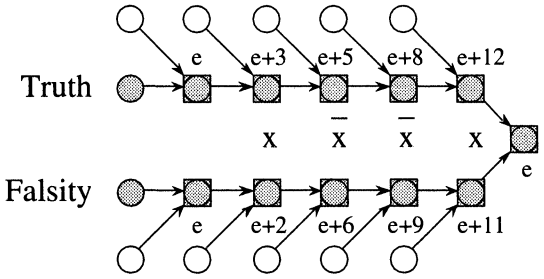


FIG. 7. *An in-tree for scheduling with deadlines on OR tasks only.*

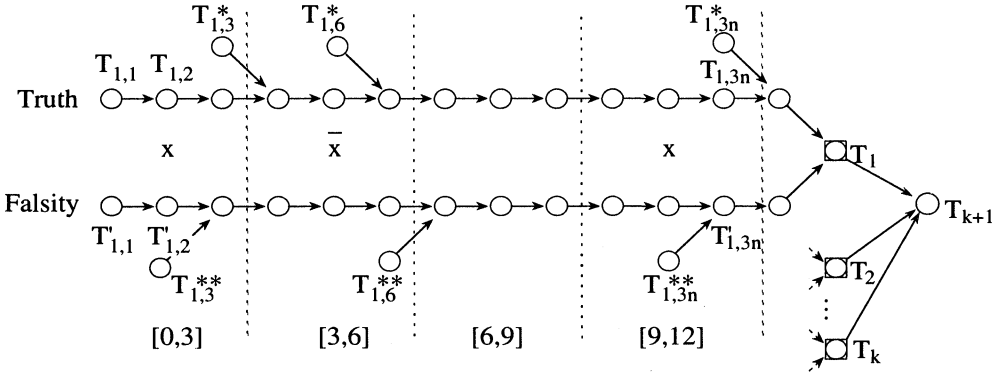


FIG. 8. *In-tree task system for AND/OR/skipped/unskipped tasks on m processors.*

THEOREM 4.3. *The problem of scheduling an AND/OR/skipped task system to minimize completion time on m processors, where tasks have identical processing times and in-tree precedence constraints, is NP-complete.*

Proof. Given a 3SAT problem with k boolean variables and n clauses, we specify a system with $m = k + 1$ processors. For each variable x_i we create an in-tree with one OR task T_i at the root and two predecessor chains of length $3n + 1$. One chain corresponds to truth and the other corresponds to falsity. All the tasks $T_1 \ldots T_k$ have a common AND direct successor T_{k+1}. For each 3SAT clause we assign an interval of three units of time starting at time zero. Hence the intervals $[0, 3]$, $[3, 6]$, $\ldots$, $[3n - 3, 3n]$ correspond to clause 1, clause 2, $\ldots$, clause n. If a variable x_i appears uncomplemented (complemented) in clause j, we create two AND tasks $T^*_{i,j}$ and $T^{**}_{i,j}$ and make their successors $T_{i,3j+1}$ and $T'_{i,3j}$ ($T_{i,3j}$ and $T'_{i,3j+1}$), respectively. Figure 8 illustrates a portion of the transformation for a 3SAT problem with $n = 4$ clauses. The variable x appears in the first, second, and fourth clauses of the 3SAT problem instance, and x is complemented in the second clause. The predecessor chains of length $3n + 1$ are used to simulate multiple deadlines, which are not allowed by the problem statement.

If a scheduling algorithm finds a feasible schedule with an overall completion time of $3n+3$, then by interchanging tasks among different processors, we can transform the schedule so that processors one through k execute a truth or falsity chain of length $3n + 1$ in the time interval $[0, 3n + 1]$, and processor $k + 1$ executes only tasks of type $T^*_{i,j}$ or $T^{**}_{i,j}$ in the same time interval. Then each task that executes in the time interval $[3j - 1, 3j]$, $1 \le j \le n$, on processor $k + 1$ corresponds to a variable or complemented variable that is true in clause j of the 3SAT problem instance. Because only one truth or falsity chain for each OR task executes

in the time interval $[0, 3n + 1]$, the truth or falsity of a variable is consistent among different 3SAT clauses. Thus, a feasible schedule can be found if and only if there is a satisfying truth assignment. □

THEOREM 3.4. *The problem of scheduling an AND/OR/unskipped task system to minimize completion time on m processors, where tasks have identical processing times and in-tree precedence constraints, is NP-complete.*

Proof. The proof is nearly identical to the proof of Theorem 4.3. Given a 3SAT problem, we generate the same in-tree as in the proof of Theorem 3.4, except we add a chain of $6n + 6$ AND successors to task T_{k+1}. Then we ask if there is a schedule with an overall completion time of $9n + 9$. In such a schedule k task chains of inessential tasks have plenty of time to complete in the time interval $[3n + 3, 9n + 9]$. It is not difficult to see that there are tasks that execute in the time intervals $[3j - 1, 3j]$, $1 \leq j \leq n$, that correspond to a satisfying truth assignment. □

Acknowledgment. We thank Mohlafi Sefika, who implemented the path-balancing heuristic and improved its description, and Sandra Broadrick-Allen, who helped improve later versions of this paper.

REFERENCES

[1] P.-R. CHANG, *Parallel algorithms and VLSI architectures for robotics and assembly scheduling*, Ph.D. thesis, Department of Electrical Engineering, Purdue University, West Lafayette, IN, 1988.

[2] E. G. COFFMAN, JR., ED., *Computer and Job Shop Scheduling Theory*, John Wiley, New York, NY, 1976.

[3] J. Y. CHUNG, J. W.-S. LIU, AND K. J. LIN, *Scheduling periodic jobs that allow imprecise results*, IEEE Trans. Comput., 39 (1990), pp. 1156–1174.

[4] E. G. COFFMAN, JR., J. Y. LEUNG, AND D. W. TING, *Bin packing: Maximizing the number of pieces packed*, Acta Inform., 9 (1978), pp. 263–271.

[5] L. S. HOMEN DE MELLO AND A. C. SANDERSON, *AND/OR graph representation of assembly plans*, IEEE Trans. Robotics Automation, 6 (1990), pp. 188–199.

[6] M. R. GAREY AND D. S. JOHNSON, *Two-processor scheduling with start-times and deadlines*, SIAM J. Comput., 6 (1977), pp. 416–428.

[7] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. H. Freeman, San Francisco, CA, 1979.

[8] M. R. GAREY, D. S. JOHNSON, B. B. SIMONS, AND R. E. TARJAN, *Scheduling unit-time tasks with arbitrary release times and deadlines*, SIAM J. Comput., 10 (1981), pp. 256–269.

[9] D. W. GILLIES AND J. W.-S. LIU, *Greed in resource scheduling*, Proc. IEEE Real-Time Systems Symposium, 10 (1989), pp. 285–294.

[10] D. W. GILLIES AND J. W.-S. LIU, *Greed in resource scheduling*, Acta Inform., 28 (1991), pp. 755–775.

[11] ———, *Scheduling Tasks with AND/OR Precedence Constraints*, Tech. report UIUCDCS-R-90-1627 (UIUC-ENG-1766), Department of Computer Science, University of Illinois, Urbana, IL, 1991.

[12] D. W GILLIES, *Algorithms to schedule tasks with AND/OR precedence constraints*, Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1993.

[13] R. L. GRAHAM, *Bounds on multiprocessing timing anomalies*, SIAM J. Appl. Math., 17 (1969), pp. 416–429.

[14] T. C. HU, *Parallel sequencing and assembly line problems*, Oper. Res., 9 (1961), pp. 841–848.

[15] E. L. LAWLER, J. K. LENSTRA, A. H. G. RINNOOY KAN, AND D. B. SHMOYS, *Sequencing and Scheduling: Algorithms and Complexity*, Tech. report BS-R8908, Department of Operations Research, Statistics, and System Theory, Centre for Mathematics and Computer Science, Amsterdam, Holland, 1989.

[16] M. C. MCELVANY, *Guaranteeing deadlines in MAFT*, Proc. IEEE Real-Time Systems Symposium, 9 (1988), pp. 130–139.

[17] J. M. MOORE, *An n job, one machine sequencing algorithm for minimizing the number of late jobs*, Management Sci., 15 (1968), pp. 102–109.

[18] D. PENG AND K. G. SHIN, *Modeling of concurrent task execution in a distributed system for real-time control*, IEEE Trans. Comput., 36 (1987), pp. 500–516.

[19] V. SALETORE AND L. V. KALE, *Obtaining first solution faster in AND and OR parallel execution of logic programs*, North American Conference on Logic Programming, 1 (1989), pp. 390–406.

SIAM J. COMPUT.
Vol. 24, No. 4, pp. 811–821, August 1995

008

WORK-PRESERVING SPEED-UP OF PARALLEL MATRIX COMPUTATIONS*

VICTOR Y. PAN† AND FRANCO P. PREPARATA‡

Abstract. Brent's scheduling principle provides a general simulation scheme when fewer processors are available than specified by the fastest parallel algorithm. Such a scheme preserves, under slow-down, the actual number of *executed* operations, also called *work*. In this paper we take the complementary viewpoint, and rather than consider the work-preserving slow-down of some fast parallel algorithm, we investigate the problem of the achievable speed-ups of computation while preserving the work of the best-known sequential algorithm for the same problem. The proposed technique, eminently applicable to problems of matrix-computational flavor, achieves its result through the interplay of two algorithms with significantly different features. Analogous but structurally different "interplays" have been used previously to improve the algorithmic efficiency of graph computations, selection, and list ranking. We demonstrate the efficacy of our technique for the computation of path algebras in graphs and digraphs and various fundamental computations in linear algebra. Some of the fundamental new algorithms may have practical value; for instance, we substantially improve the algorithmic performance of the parallel solution of triangular and Toeplitz linear systems of equations and the computation of the transitive closure of digraphs.

Key words. parallel algorithms, processor efficiency, work-optimal algorithms, linear system of equations, path algebras, paths in graphs

AMS subject classifications. 68Q22, 68Q25, 68R10

1. Introduction.

The main objective of parallel computation, which more sharply contrasts it against sequential computation, has traditionally been the minimization of computation time t, i.e., of the number of parallel steps required to solve a given problem. Another important criterion, however, is the size of the equipment, expressed as the number p of processors used in the computation. Assume, for simplicity, that a single parameter n characterizes the problem size. When t is the chosen performance criterion, frequently the resulting algorithm running in time $t(n)$ involves a very large number $p(n)$ of processors (all assumed to be identical and capable of executing one arithmetic operation in unit time). It is reasonable to assume, however, that in general the number p of available processors will not match the requirements of the fastest algorithm for a given problem instance, i.e., $p \leq p(n)$, which reflects the situation where the number of available processors is fixed and the choice of p is dictated by economic as well as engineering reasons.

Thus a typical situation is one where we have far fewer processors for use than are necessary to achieve the minimum computation time $t(n)$; this situation is dealt with by means of a version of *Brent's scheduling principle* [Br74], [KR90], which embodies a general simulation scheme of a system with $t(n)$ processors by one with a fixed number p of processors, such that

$$1 \leq p \leq p(n).$$

Specifically, let $w(n)$ denote the total number of operations actually executed by the larger (faster) system in time $t(n)$. Then the smaller (slower) system can accomplish the same task in time

$$t \leq t(n) + w(n)/p. \tag{1}$$

*Received by the editors June 9, 1992; accepted for publication (in revised form) March 4, 1994. This work was supported in part by National Science Foundation grants CCR 90-20690 and CCR-91-96152 and PSC CUNY awards 661340 and 662478. A preliminary version of this paper appeared under a different title in *Proc. 4th Annual ACM Symp. on Parallel Algorithms and Architectures* [PP92].

†Department of Mathematics and Computer Science, Lehman College, CUNY, Bronx, New York 10468 (vpan@lcvax.bitnet).

‡Computer Science Department, Brown University, Providence, Rhode Island 02912, (franco@cs.brown.edu).

A basic assumption for this simulation scheme is that the desired allocation of the p processors to their tasks may be done simply; as observed in [KR90] and illustrated below, such processor allocation is not always a straightforward matter.

This simulation scheme can be called *work-preserving slow-down* [PP92], since, while the computation is slowed down (due to the limited resources), the total amount of *actual work* $w(n)$ is preserved. Relation (1) shows that p and t are essentially inversely proportional. It also shows that the computation time is just doubled if we balance the two terms in the right-hand side of (1), i.e., if we choose $p = w(n)/t(n)$, so that for a constant penalty in computation time we may accrue a much more substantial equipment advantage. This point is illustrated by the following classical example.

Example. Summation of n numbers, $a_1, a_2, \ldots, a_n$. (For simplicity, let $\log n$ and $\log \log n$ be positive integers.) The straightforward algorithm, allotting exactly one time unit to each execution of Step 3, uses $t(n) = \log n$ steps and $n/2$ processors:

1. **begin foreach** $i = 1, \ldots, n$ **pardo** $a_{i0} := a_i$;
2. **for** $h = 1$ **to** t **do**
3. **foreach** $i = 1, \ldots, n$ **pardo** $a_{ih} := a_{2i-1,h-1} + a_{2i,h-1}$;
4. $s := a_{it}$
5. **end**

In this case, $w(n) = n - 1$. We may achieve balancing by slowing down the first $\log \log n$ steps of loop 2–3, so that we use only $n/t(n) = n/\log n$ processors at these steps. At subsequent steps, these processors are fully adequate to simulate the original system with no slow-down. Although this simulation implicitly refers to a PRAM model, the interpretation of "balancing" becomes more enlightening in the network model, where the system is a binary tree network of processors with $n/(2 \log n)$ leaves and (i) the input numbers are organized as $\log n$ wavefronts of $n/\log n$ items each, (ii) each wavefront is separately tallied by the network, and (iii) the global sum is accumulated at the root. In this case the alluded-to inverse proportionality of p and t becomes explicit if one uses a variant of Brent's principle, where the work $w(n)$ is replaced by the product $p(n) \cdot t(n)$, called "cost" in [J92]. Note that $p(n) \cdot t(n)$ is the number of executable operations if all processors are kept busy during $t(n)$ time steps, and correctly represents a cost, since it expresses the expected return on investment. If a computation C can be performed in time t with sp processors (at a cost of tsp), each executing one arithmetic operation in unit time, then C can be performed in time st with p processors, for any $s \geq 1$. This statement is readily verified by noting that each time unit of the faster system can be simulated in s time units by the slower one. Note, however, that for $p \leq p(n)$, work and cost are of the same order (see, e.g., [J92]).

Brent's principle gives a straightforward general simulation scheme that preserves the work (or cost) of some parallel computation as we decrease the number p of deployed processors. However, frequently the work $w(n)$ of the fast parallel algorithm is substantially larger than the work $w_{\text{seq}}(n)$ of the best known sequential algorithm for the same problem instance, i.e.,

$$w_{\text{seq}}(n) = o(w(n)). \tag{2}$$

Therefore, for applications for which (2) holds, it is of interest to ask the symmetric question, which is the maximum number of processors that can be deployed while maintaining the same amount of work $w_{\text{seq}}(n)$? We expect, of course, that, if (2) holds, the maximum number of deployable processors will be $o(p(n))$, where $p(n)$ is the number of processors used by the fastest algorithm. It is appropriate to call such an algorithmic technique "work-preserving speed-up," although it has been previously referred to as "supereffective slow-down," [PP92]

to contrast it against the "effective" slow-down achievable with Brent's simulation (which preserves but does not reduce the work).

As we shall show, for several important computational problems, the attainable work-preserving speed-ups, although not achieving the maximum, are still very substantial. The technique has been implicitly used in [BP90] for polynomial division and in [BP93] for computing modulo x^n the square root (and similarly the mth root for any integer $m \geq 2$) of a polynomial $p(x)$.

As we shall illustrate below, the technique involves the careful interplay of two algorithms for the given problem, which have markedly distinct features. Such a general approach—the interplay of two algorithms to achieve performance improvements—is by no means new, and has appeared in various forms in the technical literature. The earliest instance was the adoption of an "inner/outer" algorithmic structure, frequently in the context of very-large-scale-integration-circuit implementation. For example, for matrix multiplication, the outer structure is systolic (slow) and the inner structure is based on the three-dimensional mesh-of-trees (fast), i.e., a systolic algorithm involving matrix blocks, which are in turn multiplied with the fastest algorithm. Such approach provided AT^2-optimal realizations over the entire spectrum of computation times [PV80].

Another, more sophisticated instance of the methodology is the "accelerating cascade technique" proposed in [CV86]. Here two parallel algorithms are cascaded: the first is (work-) optimal but slow, the second is fast but not optimal. The objective is to use the first algorithm to reduce the problem size, so that the second algorithm can complete the task using no more than allowed for optimality. By this careful interplay, optimal work and time can be achieved for list-ranking [CV86]. The same strategy was applied in [SV81] to obtain a work/time-optimal algorithm for the maximum problem on a CRCW-PRAM. Related results on parallel graph computations are reported in [UY91], [S91a], and [S91b].

Our present approach, although belonging to this general methodology, does not adopt the slow-optimal/fast-suboptimal strategy. In a way, it is more akin to the outer/inner structure approach. Specifically, it adopts as an outer structure a sequential recursive algorithm, which typically reduces a problem of size s to a problem of size $s - 1$. If such an algorithm exists, then, when several processors are available, the idea is to make them act on larger mathematical objects (rather than single entries), to which a fast algorithm is applicable (thus providing the inner structure). The objective is to carefully balance the respective works of the inner and outer structures. In contrast to "divide-and-conquer," and, with terminological analogy, this technique could be called "shrink-and-conquer."

We want to extend the outlined approach to some fundamental computations with matrices with further application to computational problems on graphs (represented as matrices). Our study shows that the work-preserving speed-up is possible for numerous parallel matrix computations that effectively extend the solution of a problem of size s to one of size ks for any positive integers s and k.

We demonstrate our techniques for only a few matrix computational problems, in particular, for the inversion and quasi-inversion of matrices and for solving systems of linear equations. These problems are fundamental and have numerous applications to computations for linear algebra (matrix inversion), to path algebras in graphs and digraphs (quasi-inversion), and to various areas of symbolic computations (structured linear systems).

We believe that some of our algorithms have practical value. In particular, for computations in numerical linear algebra, such as solving triangular linear systems of equations (see §5, which can be read independently of §§3 and 4 and from the second half of §2), these algorithms run faster than the known customary algorithms, even when the number of processors is reasonably bounded. Furthermore, our algorithms intensively use block matrix computations, which can be effectively implemented on loosely coupled multiprocessors.

We organize our paper as follows. After some definitions and preliminaries in §§2 and 3, we show how to apply a work-preserving speed-up to quasi-inversion of matrices over the semirings and to their inversion over the fields. In §5, we treat the inversion of triangular matrices and the solution of triangular linear systems of equations. In §6 we consider the case of Toeplitz-like input matrices, having further extension to polynomial computations. Appendix A contains some auxiliary material on basic properties of Toeplitz-like matrices.

2. Definitions and preliminary results. To have a machine-independent high-level presentation, we will assume the customary PRAM models of parallel computing [KR90], [V90]. Under such models, each processor in each step performs at most one arithmetic operation. We shall adopt the "work-time framework" for evaluating algorithmic performance (see [J92]) and use the notation $O_A(t, w)$ for the pair $O(t)$ and $O(w)$, which are, respectively, simultaneous asymptotic bounds on the numbers of arithmetic steps (arithmetic parallel time) and of operations (work). It is well known that if we choose $p \leq w/t$ processors to execute the algorithm, then the number of attainable arithmetic steps is $O(w/p)$. Our algorithms do not actually depend upon choosing the PRAM model; in fact each of them consists of a simple sequence of blocks, each routinely implementable on fixed interconnections. We shall say that a parallel algorithm, with work w, is efficient if $w = \Theta(w_{\text{seq}})$, where, as above, w_{seq} is the work of the fastest known sequential algorithm for the same problem instance.

We will use some known facts about computations with matrices over the fields $\boldsymbol{F}$ and semirings $\boldsymbol{R}$ (also called dioids and path algebras). Over any field or semiring, we may compute an $m \times n$ by $n \times p$ matrix product within the following cost bounds:

$$O_A(\log n, M(m, n, p)), \tag{3}$$

where

$$M(m, n, p) \leq mnp, M(n) = M(n, n, n) \leq n^3. \tag{4}$$

Remark 1. Over the fields (and rings), we may theoretically improve this bound at least to

$$M(m, n, p) \leq mnp/(\min\{m, n, p\})^{3-\omega},$$
$$M(n) = M(n, n, n) \leq n^{\omega},$$

with $\omega < 2.376$.

The algorithms supporting the bound $\omega < 2.376$ [CW90], [P87], [BP94] are not practical, unlike some algorithms supporting the bounds $\omega < 2.81$ and even $\omega < 2.78$, which have, or promise to have, some limited practical value [GL89], [LPS92]. In the remainder of this paper, however, we shall assume that both inequalities (4) hold as equalities.

For the randomized inversion of an $n \times n$ matrix over a field F, we have the following estimates [P91], [P92], [KP91], [KP92], [BP94]:

$$O_A(\gamma(n, q) \log^2 n, M(n) \log n), \tag{5}$$

where $M(n)$ is defined by (3), (4), q is the characteristic of F, and

$$\gamma(n, q) = \begin{cases} \lceil \log n / \log q \rceil & \text{if } q > 0, \\ 1 & \text{if } q = 0. \end{cases} \tag{6}$$

For the same computational problem, we have the following deterministic estimates:

$$O_A(\log^2 n, n^{\beta} M(n)), \tag{7}$$

where $\beta \leq 1$ over any field F [Be84], [Ch85]; however, if F has characteristic $q = 0$ or $q > n$, then $\beta < 1/2$ [Cs76], [PS78], [GP89].

Over the semirings with additive operator $\oplus$, instead of matrix inverses, we compute the quasi-inverses (A^* denoting the quasi-inverse of A), as follows:

$$A^* = \lim_{h\to\infty} A^{(h)}, \quad A^{(0)} = I, \quad A^{(h)} = A^{(h-1)} \oplus A^h = I \oplus A \oplus \ldots \oplus A^h, \quad h = 1, 2, \ldots,$$

where I denotes the identity matrix in the semiring [Ca79], [PR89], [P93]. In many applications, we deal with semirings where

$$A^* = A^{(n-1)} = \prod_{j=0}^{\lceil \log(n-1)\rceil - 1} (I + A^{2^j}), \tag{8}$$

for any $n \times n$ matrix A. Then the bounds (3), (4) for $M(n) = n^3$ are extended to the evaluation of A^*; these bounds take the form

$$O_A(\log^2 n, n^3 \log n), \tag{9}$$

thus yielding a parallel algorithm that is fast but not work-optimal, since in this case $w_{\text{seq}} = O(n^3)$ (see, e.g., [AHU74]).

3. A factorization of a matrix and its (quasi-) inverse. Hereafter, I and 0 denote the identity and the null matrices of appropriate sizes, respectively. A^T and A^H denote the transpose and the Hermitian transpose of a matrix A, respectively (see [GL89] for these standard definitions).

We will represent an $n \times n$ matrix A as a 2×2 block matrix and recall the following factorizations of A and A^{-1} [AHU74]:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \qquad A = \begin{bmatrix} I & 0 \\ A_{21}A_{11}^{-1} & I \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} I & A_{11}^{-1}A_{12} \\ 0 & I \end{bmatrix}, \tag{10}$$

$$A^{-1} = \begin{bmatrix} I & -A_{11}^{-1}A_{12} \\ 0 & I \end{bmatrix} \begin{bmatrix} A_{11}^{-1} & 0 \\ 0 & S^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -A_{21}A_{11}^{-1} & I \end{bmatrix}, \tag{11}$$

$$S = A_{22} - A_{21}A_{11}^{-1}A_{12}. \tag{12}$$

Here, A_{11} is an $m \times m$ matrix, $m \leq n$, and we assume nonsingularity of A, A_{11}, and S. (S is known as the Schur complement of A_{11} in A). For a nonsingular matrix A, we may ensure nonsingularity of A_{11} and S, with a high probability, by means of a simple preconditioning of A, for instance, by the transition to the matrix UAL^T, where U and L^T are unit lower triangular Toeplitz matrices with ones on their diagonals and with other entries of their first columns chosen at random (compare [KP91], [BP94]). Over the real or complex fields (and over their subfields) we may deterministically ensure the nonsingularity of A_{11} and S by shifting from the original matrix A to the positive definite product A^HA.

No extension of (10) to semirings is known, but (11) and (12) are extended as follows ([AHU74, p. 205], [PR89], [P93]):

$$A^* = \begin{bmatrix} I & A_{11}^*A_{12} \\ 0 & I \end{bmatrix} \begin{bmatrix} A_{11}^* & 0 \\ 0 & S^* \end{bmatrix} \begin{bmatrix} I & 0 \\ A_{21}A_{11}^* & I \end{bmatrix}, \tag{13}$$

$$S = A_{22} + A_{21}A_{11}^*A_{12}. \tag{14}$$

4. Work-preserving speed-up for matrix inversion and quasi-inversion. The fastest parallel algorithms that support the bounds (5), (7), and (9) are not efficient since Gaussian elimination over any field and its extensions to semirings [Ca79], [PR89] support the bounds $O_A(n, n^3)$. Next, we will demonstrate that matrix inversion and quasi-inversion lend themselves to work-preserving speed-ups.

Let us consider a semiring such that property (8) holds for $n \times n$ matrices for any n (this implies that (9) is applicable). Let A be an $n \times n$ matrix. As in the preceding section, we partition A as a 2×2 block matrix, where the upper-left block A_{11} is an $h \times h$ matrix (h to be determined). The technique based on (13) and (14) reduces the computation of A^* to the computation of A_{11}^*, S^*, and six multiplications of pairs of rectangular matrices, whose dimensions never exceed n and one of which has at least one dimension equal to h. Therefore, these multiplications have global performance $O_A(\log n, n^2 h)$ (see (3) and (4)). If we explicitly compute A_{11}^* and all matrix products we reduce the original problem to the (recursive) computation of S^*. By (8), A_{11}^* is computed at the cost $O_A(\log^2 h, h^3 \log h)$. Thus, the nonrecursive part of the computation satisfies the cost bound

$$O_A(\max(\log^2 h, \log n), \max(h^3 \log h, n^2 h)). \tag{15}$$

It follows that the performance of the global computation is obtained by multiplying by n/h each of the terms of (15). Next, we impose the condition $h^3 \log h \leq n^2 h$, which yields $\max(h^3 \log h, n^2 h) = n^2 h$. If, specifically, we select $h = n/(\log n)^{1/2}$ (which satisfies the preceding condition), then we have $(n/h)\max(\log^2 h, \log n) = (n/(n/(\log n)^{1/2}))\log^2 n = \log^{5/2} n$ and $(n/h)\max(h^3 \log h, n^2 h) = (n/h)n^2 h = n^3$, so that we arrive at the solution A^* of the original problem with the following (work-optimal) performance:

$$O_A((\log n)^{5/2}, n^3). \tag{16}$$

The parallel algorithm that supports (15) is efficient, according to our definition, and still quite fast, for it fails to attain just by a factor $O((\log n)^{1/2})$ the time of the fastest known algorithm for this problem (see (9)).

Over the fields of characteristic q, a similar approach based on (5), (10)–(12) [where we choose $h = n/(\log n)^{1/2}$ (see (5)) as the dimension of block A_{11} and where we precondition A to avoid the singularities] yields the bounds

$$O_A((\log n)^{5/2}\gamma(n, q), n^3)$$

on the randomized complexity of the inversion of an $n \times n$ matrix.

When we operate over the fields of characteristic 0, we deterministically ensure nonsingularity in all such recursive computations by using $A^H A$ in lieu of A (since A^{-1} is easily obtained through the identity $A^{-1} = (A^H A)^{-1} A^H$) (compare [BP94]). Then we may also apply (7) (in this case with $\beta = 1/2$), and (10)–(12). If we choose $h = n^{4/5}$, the same techniques enable us to compute the inverse A^{-1} of an $n \times n$ nonsingular matrix A with the performance

$$O_A(n^{1/5} \log^2 n, n^3). \tag{17}$$

Note that (7), and consequently (17), are deterministic bounds. Again, we obtain an efficient algorithm, although in this case the speed-up falls substantially short of the maximum (the attainable time is $n^{1/5} \log^2 n$, rather than $\log^2 n$).

Remark 2. The latter estimate relies on (7) with $\beta = 1/2$. For $\beta < 1/2$, we may decrease the time bound, preserving the work n^3.

As an exercise, the reader may work out the extension of (15) and (16) based on the estimates of Remark 1. Then the asymptotic estimate for the work will decrease but will not reach the bound h^{ω} since (according to equations (10)–(12)) this involves rectangular matrix multiplication, which generally is not asymptotically as fast as square matrix multiplication (refer to Remark 1).

5. Triangular matrix inversion and solution of triangular linear systems. Let A be an $n \times n$ nonsingular triangular matrix. Then we may improve the estimates of the previous section for the inversion of A as follows [BM75, p. 146]:

$$O_A(\log^2 n, n^3). \tag{18}$$

Indeed, represent A and A^{-1} as 2×2 block matrices

$$A = \begin{bmatrix} A_{11} & 0 \\ A_{21} & A_{22} \end{bmatrix}, \qquad A^{-1} = \begin{bmatrix} A_{11}^{-1} & 0 \\ -A_{22}^{-1}A_{12}A_{11}^{-1} & A_{22}^{-1} \end{bmatrix}. \tag{19}$$

Assume for convenience that $n = 2^k$, for an integer k, and that A_{11} is an $(n/2) \times (n/2)$ block and note that (19) reduces the inversion of A to the concurrent inversions of the half-size matrices A_{11} and A_{22} and to two subsequent multiplications of $(n/2) \times (n/2)$ matrices. Recursively apply the same observations to invert A_{11} and A_{22}. Using (3), (4), and Brent's principle, we obtain a fast and efficient $O_A(\log^2 n, n^3)$ algorithm.

We may now use this algorithm to achieve a work-preserving speed-up for the solution of the triangular linear systems

$$A\mathbf{x} = \mathbf{b}, \tag{20}$$

since $\mathbf{x} = A^{-1}\mathbf{b}$. Note that the computation of A^{-1} is not required to obtain $\mathbf{x}$; if we first compute A^{-1} to subsequently obtain $\mathbf{x}$, the resulting algorithm is not efficient, since the simple (forward or back) substitution algorithm yields the bounds $O_A(n, n^2)$. We next explore whether a work-preserving speed-up is achievable for this problem.

We assume now that A_{11} is an $h \times h$ matrix (again, h is a parameter to be selected) and let $\mathbf{b}^T = [\mathbf{b}_1^T, \mathbf{b}_2^T]$, where $\mathbf{b}_1$ is a vector of dimension h. From (19) we deduce that

$$\mathbf{x} = A^{-1}\mathbf{b} = \begin{bmatrix} \mathbf{e} \\ A_{22}^{-1}\mathbf{f} \end{bmatrix}, \quad \mathbf{e} = A_{11}^{-1}\mathbf{b}_1, \quad \mathbf{f} = -A_{12}\mathbf{e} + \mathbf{b}_2. \tag{21}$$

These relations suggest to compute $\mathbf{e}$ at the cost $O_A(\log^2 h, h^3)$ (see (18)) and subsequently to compute $\mathbf{f}$ at the cost $O_A(\log h, nh)$. By so doing, we have reduced the original problem to the problem of computing $A_{22}^{-1}\mathbf{f}$, that is, to solving the triangular linear system $A_{22}\mathbf{y} = \mathbf{f}$ of $n-h$ equations. Iterating this approach, we could successively reduce this system to triangular systems of $n-2h, n-3h, \ldots$ equations, and thereby solve the original system (20). However, to avoid the inefficiency deriving from the imbalance between the computation times of $\mathbf{e}$ and $\mathbf{f}$ given above ($O(\log^2 h)$ vs. $O(\log h)$), we propose the following more efficient strategy. The computation is carried out as a sequence of $n/(h \log h)$ major iterations, each consisting of $\log h$ minor iterations, essentially as described above, the only difference being that the inversions of the $\log h$ diagonal blocks are carried out, concurrently, at the beginning of the major iterations, at the cost $O_A(\log^2 h, h^3 \log h)$. It follows that the overall cost is

$$O_A\left(\frac{n}{h \log h} \log^2 h, \frac{n}{h \log h} \max\left(h^3 \log h, nh \log h\right)\right),$$

which leads to selecting $h = n^{1/2}$ and to the estimate

$$O_A(n^{1/2}\log n, n^2). \tag{22}$$

Note that the algorithm is efficient (the work equals n^2), but we do not know how to avoid inefficiency if t is polylogarithmic in n or even if $t = n^{1/4}$ (for example).

Remark 3. We may obtain (theoretical) asymptotic improvements of the bounds (22) by applying the estimates of Remark 1. Then we just need to set $h = n^{1/(\omega-1)}$ and arrive at the bounds $O_A(n^\alpha \log^2 n, n^2)$, $\alpha = 1 - 1(\omega - 1)$, so that $\omega < 2.378$ implies that $\alpha < 0.28$. Clearly, a polylogarithmic time bound is achieved only for $\alpha = 0$ implied by the value $\omega = 2$.

6. Solving structured linear systems and further applications. In this section we will refer to computations over the complex field of constants, so that fast Fourier transform (FFT) on m points can be performed at the cost $O_A(\log m, m \log m)$. We will show a work-preserving speed-up for solving a linear system (20) in the special case where A denotes an $n \times n$ Toeplitz or Toeplitz-like matrix represented by its displacement generator of length $O(1)$.

We refer to Appendix A for the definition and basic properties of Toeplitz-like matrices and their displacement generators. By using these representations and their properties, one may devise two algorithms that compute a short displacement generator of the inverse of a nonsingular $n \times n$ Toeplitz-like matrix A at the cost bounded by $O_A(n, n\log^2 n)$, [BA80], [M80], and $O_A(\log^2 n, n^2 \log n)$ [P92], respectively.

Then, after this preprocessing stage and independent of the vector $\mathbf{b}$, the solution $\mathbf{x} = A^{-1}\mathbf{b}$ to the linear system $A\mathbf{x} = \mathbf{b}$ can be immediately computed at the cost $O_A(\log n, n \log n)$. Adopting the latter of the two cited algorithms, one should proceed as in §4, noting, however, that the matrix $A^H A$, as well as the matrices A_{11}, A_{11}^{-1}, A_{12}, A_{21}, A_{22}, and S of (10)–(12), are now Toeplitz-like matrices, to be represented in terms of their displacement generators of lengths $O(1)$. Much less computational work is involved when operating with such matrices than when dealing with general matrices as in §4.

In this way, one will finally obtain an algorithm that computes $A^{-1}\mathbf{b}$ within the bounds

$$O_A(n^{1-a}\log^2 n, n^{1+a}\log n), \tag{23}$$

for any a, $1/2 \le a \le 1$. For instance, (23) turns into $O_A(n^{1/2}\log^2 n, n^{3/2}\log n)$ for $a = 1/2$. To achieve this result, for a given a, one should choose as $n^a \times n^a$ the size of the leading block A_{11} in (10)–(12), thereby obtaining a Schur complement S of A_{11} in A of size $(n - n^a) \times (n - n^a)$ at the cost $O_A(\log^2 n, n^{2a}\log n)$, and, similarly, for each of the subsequent n^{1-a} iterations. Note, however, that one never explicitly computes the n^2 entries of A^{-1} (of which $(n + 1)n/2$ may be distinct even when A is a Toeplitz matrix), but only recursively expresses the displacement generators of A^{-1} via those of A_{11}^{-1} and S^{-1}; furthermore, S^{-1} is a trailing principal submatrix of A^{-1}, and similarly are the Schur complements in S (and in its trailing blocks), arising in the outlined process of the recursive factorization of A^{-1}.

These bounds can be immediately extended to the solution of linear systems with $n \times n$ Hankel-like, Hilbert-like, and Vandermonde-like matrices (see [P90]), to the evaluation of greatest common divisor and least common multiple of two polynomials of degrees $O(n)$ (see [P92]), and to several other fundamental computations with matrices and polynomials (see [BP94]).

Appendix A. Some basic properties of Toeplitz and Toeplitz-like matrices. In this appendix we recall some basic properties of Toeplitz and Toeplitz-like matrices (compare [KKM79], [CKL87], [BP94], [P90], [P92], [P92a]).

The $n \times n$ matrix $Z = [z_{i,j}]$, which is 0 everywhere except for all 1's on the first subdiagonal, is called the $n \times n$ lower shift (displacement) matrix. (Note that $Z^n = 0$.) Z

generates the algebra of $n \times n$ lower triangular Toeplitz matrices, $\{\sum_{i=0}^{n-1} a_i Z^i = L(\mathbf{a})\}$, and similarly, Z^T generates the algebra of $n \times n$ upper triangular Toeplitz matrices,

$$\left\{ L^T(\mathbf{b}) = \sum_{i=0}^{n-1} b_i (Z^T)^i \right\},$$

where $\mathbf{a} = (a_0, \ldots, a_{n-1})^T$ and $\mathbf{b} = (b_0, \ldots, b_{n-1})^T$. An $n \times n$ Toeplitz matrix $T = [t_{i,j}]$ is a matrix representable as the sum $L(\mathbf{a}) + L^T(\mathbf{b})$ for any fixed pair of vectors $\mathbf{a}$ and $\mathbf{b}$. (Note that $t_{i,j} = t_{i+k,j+k}$ for any triple i, j, k, where $i, j, i+k, j+k$ are within the range $1, 2, \ldots, n$.) More generally, a pair $G = [\mathbf{g}_1, \ldots, \mathbf{g}_d]$, $H = [\mathbf{h}_1, \ldots, \mathbf{h}_d]$ of $n \times d$ matrices generates the matrices

$$A = \sum_{i=1}^{d} L(\mathbf{g}_i) L^T(\mathbf{h}_i), \tag{A.1}$$

$$B = \sum_{i=1}^{d} L^T(\mathbf{g}_i) L(\mathbf{h}_i). \tag{A.2}$$

The pair (G, H) is called either the (ZZ^T)-displacement generator of length d for A or the (Z^TZ)-displacement generator of length d for B. (Note the simple transition from (A.1), (A.2) to the displacement generators for A^H, B^H). For a given A, the smallest possible length d^* of its displacement generators is called the displacement rank of A.

THEOREM A.1 [KKM79]. *Equation* (A.1) *holds if and only if* $A - ZAZ^T = GH^T$; (A.2) *holds if and only if* $B - Z^TBZ = GH^T$.

By Theorem A.1, matrices can be represented by their displacement generators, which reduces the storage requirements if the generators are short (i.e., of small length). Moreover, recalling that multiplication of a Toeplitz matrix by a vector reduces to polynomial multiplication (convolution of vectors) and can be performed at the cost $O_A(\log n, n \log n)$, one can multiply either A or B, given by (A.1) and (A.2), by a vector at the cost $O_A(\log n, dn \log n)$ (since each such multiplication is resolved into $2d$ multiplications of a Toeplitz matrix by a vector). If $d = O(1)$, then A and B are called "Toeplitz-like."

The sum, difference, and product of two Toeplitz-like matrices U and V are Toeplitz-like matrices, whose displacement generators are explicitly expressed via the homologous generators of U and V. Moreover, similar properties hold for all the leading and trailing principal submatrices of U and V. Finally, if $UV = I$, then the displacement rank of U equals the displacement rank of V [KKM79], and for a Toeplitz-like matrix A represented with its displacement generator (G, H) according to (A.1), one may effectively compute its shortest ZZ^T- and Z^TZ-displacement generators, and similarly for the matrix B represented according to (A.2) [BA80], [P92], [P93a].

Acknowledgment. The authors wish to thank David Eppstein for his interest in the subject of this paper and for his useful comments.

REFERENCES

[AHU74] V. AHO, J. E. HOPCROFT, and J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[BA80] R. R. BITMEAD AND B. D. O. ANDERSON, *Asymptotically fast solution of Toeplitz and related systems of linear equations*, Linear Algebra Appl., 41 (1980), pp. 111–130.

[Be84] S. BERKOWITZ, *On computing the determinant in small parallel time using a small number of processors*, Inform. Process. Lett., 18 (1984), pp. 147–150.

[Br74] R. P. BRENT, *The parallel evaluation of general arithmetic expressions*, J. Assoc. Comput. Mach., 21 (1974), pp. 201–206.

[BM75] A. BORODIN AND I. MUNRO, *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, New York, 1975.

[BP90] D. BINI AND V. Y. PAN, *Parallel polynomial computations by recursive processes*, in Proc. ACM SIGSAM Intern. Symp. on Symb. and Alg. Comp. (ISSAC-90), ACM Press, NY, 1990, p. 294.

[BP93] ———, *Improved parallel polynomial division*, SIAM J. Comput., 22 (1993), pp. 617–626.

[BP94] ———, *Polynomial and Matrix Computations* I, Birkhauser, Boston, 1994.

[Ca79] B. A. CARRÉ, *Graphs and Networks*, The Claredon Press, Oxford, 1979.

[Ch85] A. L. CHISTOV, *Fast parallel calculation of the rank of matrices over a field of arbitrary characteristics*, in Proc. FCT 85, Springer Lecture Notes in Computer Science, 199, Springer-Verlag, Boston, 1985, pp. 63–69.

[Cs76] L. CSANKY, *Fast parallel matrix inversion algorithms*, SIAM J. Comput., 5 (1976), pp. 618–623.

[CKL87] J. CHUN, T. KAILATH, AND H. LEV-ARI, *Fast parallel algorithm for QR-factorization of structured matrices*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. 899–913.

[CV86] R. COLE AND U. VISHKIN, *Deterministic coin tossing with applications to optimal parallel list ranking*, Inform. and Control, 70 (1986), pp. 32–53.

[CW90] D. COPPERSMITH AND S. WINOGRAD, *Matrix multiplication via arithmetic progression*, J. Symbolic Comput., 9 (1990), pp. 251–280.

[GL89] G. H. GOLUB AND C. F. VAN LOAN, *Matrix Computations*, Johns Hopkins University Press, Baltimore, 1989.

[GP89] Z. GALIL AND V. Y. PAN, *Parallel evaluation of the determinant and of the inverse of a matrix*, Inform. Proc. Lett., 30 (1989), pp. 41–45.

[J92] J. JÀJÀ, *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA, 1992.

[KKM79] T. KAILATH, S.-Y. KUNG, AND M. MORF, *Displacement rank of matrices and linear equations*, J. Math. Anal. Appl., 68 (1979), pp. 395–407.

[KP91] E. KALTOFEN AND V. Y. PAN, *Processor efficient parallel solution of linear systems over an abstract field*, in Proc. 3rd Annual ACM Symp. on Parallel Algorithms and Architecture, Hilton Head, SC, ACM Press, New York, 1991, pp. 180–191.

[KP92] ———, *Parallel and processor efficient solution of linear systems* II: *The general case*, in Proc. 33rd IEEE Symposium on Foundations of Computer Science, Pittsburgh, PA, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 714–723.

[KR90] R. KARP AND V. RAMACHANDRAN, *A survey of parallel algorithms for shared memory machines*, in Handbook of Theoretical Computer Science, North-Holland, Amsterdam, 1990, pp. 869–941.

[LPS92] J. LADERMAN, V. Y. PAN, AND X.-H. SHA, *On practical acceleration of matrix multiplication*, Linear Algebra Appl., 162–164 (1992), pp. 557–588.

[M80] M. MORF, *Doubling algorithms for Toeplitz and related equations*, in Proc. IEEE Intern. Conference on ASSP, IEEE Computer Society Press, 1980, pp. 954–959.

[P87] V. Y. PAN, *Complexity of parallel matrix computations*, Theoret. Comput. Sci., 54 (1987), pp. 65–85.

[P90] ———, *On computations with dense structured matrices*, Math. of Comput., 55 (1990), pp. 179–190.

[P91] ———, *Complexity of Algorithms for Linear Systems of Equations*, in Computer Algorithms for Solving Linear Algebraic Equations (The State of the Art), edited by E. Spedicato, NATO ASI Series, Series F: Computer and Systems Sciences, Springer-Verlag, Berlin, 77, 1991, pp. 27–56.

[P92] ———, *Parameterization of Newton's iteration for computations with structured matrices and applications*, Comput. Math. Appl., 24 (1992), pp. 61–75.

[P92a] ———, *Complexity of computations with matrices and polynomials*, SIAM Rev., 34 (1992), pp. 225–262.

[P93] ———, *Parallel solution of sparse linear and path systems*, in Synthesis of Parallel Algorithms, J. Reif, ed., Morgan Kaufmann, San Mateo, CA, 1993, pp. 621–678.

[P93a] ———, *Decreasing the displacement rank of a matrix*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 118–121.

[PP92] V. Y. PAN AND F. P. PREPARATA, *Supereffective slow-down of parallel computations*, in Proc. 4th Annual ACM Symp. on Parallel Algorithms and Architectures, San Diego, CA, ACM Press, New York, 1992, pp. 402–409.

[PR89] V. Y. PAN AND J. REIF, *Fast and efficient solution of path algebra problems*, J. Comput. System Sci., 38 (1989), pp. 494–510.

[PS78] F. P. PREPARATA AND D. V. SARWATE, *An improved parallel processor bound in fast matrix inversion*, Inform. Proc. Lett. 7 (1978), pp. 148–149.

[PV80] F. P. PREPARATA AND J. VUILLEMIN, *Area-time optimal VLSI networks for multiplying matrices*, Inform. Process. Lett., 11 (1980), pp. 77–80.

[S91a] T. SPENCER, *Time-work tradeoffs for parallel graph algorithms*, in Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms, San Francisco, CA, 1991, ACM Press, New York, and Society for Industrial and Applied Mathematics, Philadelphia, PA, pp. 425–432.

[S91b] ———, *More time-work tradeoffs for parallel graph algorithms*, in Proc. 3rd Annual ACM Symp. on Parallel Algorithms and Architectures, Hilton Head, SC, 1991, ACM Press, New York, pp. 81–93.

[SV81] Y. SHILOACH AND U. VISHKIN, *Finding the maximum, merging, and sorting in a parallel computation model*, J. Algorithms, 2 (1981), pp. 88–102.

[V90] L VALIANT, *General purpose parallel architectures*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., North Holland, Amsterdam, (1990), pp. 943–971.

[UY91] J. ULLMAN AND M. YANNAKAKIS, *High-probability parallel transitive closure algorithms*, SIAM J. Comput., 20 (1991), pp. 100–125.

SIAM J. COMPUT.
Vol. 24, No. 4, pp. 822–839, August 1995

009

INTEGER LINEAR PROGRAMS AND LOCAL SEARCH FOR MAX-CUT*

SVATOPLUK POLJAK†

Abstract. The paper deals with the complexity of the local search, a topic introduced by Johnson, Papadimitriou, and Yannakakis. One consequence of their work, and a recent paper by Schäffer and Yannakakis, is that the local search does not provide a polynomial-time algorithm for finding locally optimum solutions for several hard combinatorial optimization problems. This motivates us to seek "easier" instances for which the local search is polynomial. In particular, it has been proved recently by Schäffer and Yannakakis that the max-cut problem with the FLIP neighborhood is polynomial-time local search (PLS) complete, and hence belongs among the most difficult problems in the PLS class. The FLIP neighborhood of a 2-partition is defined by moving a single vertex to the opposite class. We prove that, when restricted to cubic graphs, the FLIP local search becomes "easy" and finds a local max-cut in $O(n^2)$ steps. To prove the result, we introduce a class of integer linear programs associated with cubic graphs and provide a combinatorial characterization of their feasibility.

Key words. local search, local optima, max-cut, cubic graph, integer linear program

AMS subject classifications. 68Q25, 90C10, 90C27

1. Introduction.

Local search is a frequently used practical strategy for finding good feasible solutions for hard combinatorial optimization problems. Starting from an arbitrary initial solution, a sequence of feasible solutions is iteratively generated so that a consecutive feasible solution is always chosen in the neighborhood of the current one, and has a strictly better cost. When no improvement within the neighborhood is possible, a *local optimum* is found. Some well-known examples of local search are the Lin–Kernighan heuristic for graph partitioning, 2-exchange for the traveling salesman problem, and the simplex method of linear programming.

Very often the local search needs only a few number of steps to terminate, but the number of steps is exponential in the worst case for most problems.

The theoretical importance of the local search was recognized by Johnson, Papadimitriou, and Yannakakis [4], who introduced a complexity class *polynomial-time local search* (PLS) in order to capture the essence of the local search and study its complexity. An important concept is that of a *neighborhood structure*, because one can consider several different neighborhood structures for the same optimization problem. For example, 2-exchange and 3-exchange are different neighborhoods for the traveling salesman problem, and they also lead to different local optima in general. For the purpose of our paper we do not need to present the detailed definition of the PLS class. Let us only recall that a problem P belongs to the class PLS if a triple of polynomial-time algorithms is given: (i) an algorithm generating an initial feasible solution, (ii) an algorithm computing an integral cost of a given feasible solution, and (iii) an algorithm which, given a feasible solution, either finds a better neighboring solution or determines that the current solution is a local optimum.

A crucial question posed in [4] was whether there is a polynomial-time algorithm for finding a local optimum for a PLS problem. Local search itself does not provide such an algorithm since the number of its steps need not be polynomially bounded. On the other hand, interior point algorithms for linear programming show that local search is not the only option for finding a local optimum (which is also global in case of the linear programming) and, morever, provide a polynomial-time algorithm.

*Received by the editors March 10, 1993; accepted for publication (in revised form) March 14, 1994.

†University of Passau, Faculty of Mathematics and Informatics, Innstrasse 33, 94030 Passau, Germany, until his death in 1995. This research was done while the author visited the Institute of Mathematics of Academia Sinica in Taipei, and was supported by grant VS92021 of the National Science Council of the Republic of China.

The main results of [4] were that the class PLS contains *complete* problems, which include both interesting and practical problems. (A problem P in PLS is said to be complete if the existence of a polynomial-time algorithm for finding local optima for the instances of P implies the existence a polynomial-time algorithm for all problems in PLS.) This first PLS-complete problem was the graph partitioning under the Kerninghan–Lin neighborhood. Later, several other PLS-complete problems were found [6], [10], [8]. However, the question of the complexity of finding local optima remains unsolved, and there are several arguments indicating that its complexity may lie "somewhere between" the polynomial-time and the NP-complete problems (cf. [4], [9]).

Our paper is motivated by an opposite goal, namely, the following question: Which problems in the class PLS are *easy*? Following a certain analogy with the P=NP? problem, one can say that the investigation of polynomial-time solvable problems gained in significance simply because of the theory of NP-completness, and it is also considered important to find polynomial-time algorithms for restricted versions of NP-hard problems.

We study the problem of locally maximum cuts in weighted graphs. The *FLIP neighborhood* of a cut consists of the cuts obtained by moving a single vertex in the opposite class of the 2-partition. Schäffer and Yannakakis [10] proved that the problem of local max-cut with the FLIP neighborhood is PLS-complete. An immediate consequence is that FLIP local search requires an exponential number of steps in the worst case. Our main result is that the FLIP local search becomes easy when restricted to cubic graphs. We prove that the FLIP local search cannot be longer than $6n^2$ steps for a weighted cubic graph on n vertices. This contrasts with the optimization version—the max-cut problem, which remains NP-complete even when restricted to cubic graphs (see [11]). The case of cubic graphs was first studied by Loebl [7], who proved that a FLIP local max-cut can be found in polynomial time for a cubic graph with nonnegative edge weights. However, his algorithm avoids local search, and he conjectured (personal communication) that local search might be exponential even in cubic graphs. Now the existence of a polynomial-time algorithm immediately follows from our result since the local search itself is polynomial. Another extension of [7] is that our result is valid for *arbitrary* edge weights, while Loebl [7] considered only nonnegative weights.

Loebl [7] claims that his algorithm can be implemented in linear time. Let us remark that a linear-time algorithm can also be obtained from the analysis of our paper.

A natural question is whether the FLIP local search also remains easy for the graphs with bounded maximum degree D for $D > 3$. I have been informed by one of the referees that this is not true. He remarked that a consequence of the result of Krentel [5] is that there exits a degree D_0 such that for every $D \geq D_0$, there is an infinite family of weighted graphs of maximum degree D and corresponding initial 2-partitions, such that *every* local search from the initial solution takes exponential time. A related result which already shows an increased complexity for 4-regular graphs (with edge- and node-weights) is by Haken and Luby [3]. They presented an infinite family of 4-regular graphs together with initial 2-partitions for which the *greedy* local search (FLIP the vertex that yields the most improvement) is exponential. Thus, the results of [5] and [3] indicate that FLIP local search and FLIP local max-cut might already be difficult for 4-regular graphs, but the problem remains open. On the other hand, the method of our paper does not seem to allow extension to 4-regular graphs.

Let us briefly explain the approach of our paper. Assume we are given a cubic graph with nonnegative integer edge weights c_e, $e \in E$. It is easy to see that any FLIP local search is bounded by the total sum of all edge weights since the weight of the cut is strictly increasing in each step of the local search. However, $\sum_{e \in E} c_e$ does not provide a polynomial upper bound on the local search since it can be exponential in the size of the binary encoding of G and c. Our goal is to show that every c can be replaced by some other *bounded integer* function

w for which the FLIP local search runs precisely the same way as for c. That is, we will construct w such that every w_e, $e \in E$, is bounded by $O(n)$. Since there are $O(n)$ edges, the total edge weight is $O(n^2)$. Every step of local search strictly improves the weight of the current solution, and thus the maximum possible length of a sequence of local search is at most $O(n^2)$. The order of the polynomial n^2 is the best possible. We present an example of an infinite family of cubic graphs that admit $\binom{n}{2}$ steps of local search, which establishes that local search in cubic graphs is of order $\Theta(n^2)$.

The new edge weights w are found as a solution of a system of linear inequalities in variables x_e, $e \in E$, which is determined by c. We call such a system of linear inequalities a *local linear program* since the constraints are associated with the vertices of the graph, and every variable x_{ij} occurs only in the constraints corresponding to i or j. The following ideas lead to the definition of a local linear program. A vertex i is said to be *c-happy* in some 2-partition if flipping i does not increase the weight of the cut. Let e, f, and g denote the triple of edges incident to a vertex i. The "happiness" of a vertex i depends on the mutual inequalities between c_e, c_f, and c_g rather than on their actual values. We observe that only two cases have to be distinguished. Either (i) none of $\{c_e, c_f, c_g\}$ is larger than the sum of the remaining two, or (ii) one of $\{c_e, c_f, c_g\}$ is strictly larger than the sum of the remaining two. In the former case, a vertex is happy if and only if at least two of its edges are in the cut. In the latter case, a vertex is happy if and only if the "heaviest" edge is in the cut.

Thus, constraints (i) and (ii) provide necessary and sufficient conditions for w to be equivalent with c. However, it is not immediately obvious how to construct w so that it is bounded. We proceed as follows: It appears convenient to "forget" about the original weights c and study only the system of linear inequalities determined by it. That means that we introduce an *abstract* local linear program not necessarily associated with any edge weight function, and hence possibly *infeasible*. We provide a combinatorial characterization of the feasibility of the (abstract) local linear programs and, as a by-product, we conclude that if a local linear program is feasible, then it has a bounded integral solution. The characterization of the feasibility is formulated in terms of the mutual relation between the structure of the underlying graph and the linear constraints associated with the vertices of the graph.

The paper is organized as follows: The notion of a local linear program is introduced in §2, where we also formulate the result on the existence of bounded integer solutions. Section 3 is devoted to the main application, the FLIP local search in cubic graphs. In §4 we provide a combinatorial characterization of local linear programs associated with cubic graphs. A consequence of this characterization is the existence of the bounded integer solutions for feasible local linear programs.

2. Local linear programs. We consider graphs without loops and multiple edges. A graph is called *cubic* if the degree of every vertex is three. Let $G = (V, E)$ be a cubic graph on n vertices, where $V = \{1, \ldots, n\}$. Let $x = (x_e)$, $e \in E$, be a vector of variables associated with the edges of G. A system of inequalities

$$A_i x \geq b_i \tag{1}$$

is called *a block* of vertex i if it has one of the forms (2) or (3).

Let e_1, e_2, and e_3 be the triple of edges adjacent to i, and let x_{e_1}, x_{e_2}, and x_{e_3} be the corresponding variables. A *block associated with vertex i* is either a system of three inequalities

$$\begin{aligned} x_{e_1} + x_{e_2} - x_{e_3} &\geq 0, \\ x_{e_1} - x_{e_2} + x_{e_3} &\geq 0, \\ -x_{e_1} + x_{e_2} + x_{e_3} &\geq 0, \end{aligned} \tag{2}$$

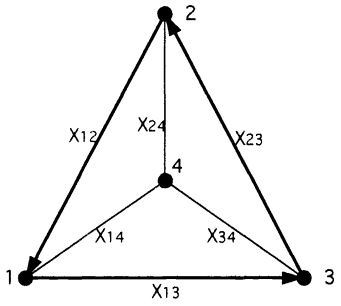


FIG. 1. *A local linear program associated with* K_4.

or just one inequality

$$x_{e_j} - x_{e_k} - x_{e_\ell} \geq 1, \quad \text{where } \{j, k, \ell\} = \{1, 2, 3\}. \tag{3}$$

In other words, the meaning of (2) is that none of $x_{e_1}, x_{e_2}, x_{e_3}$ is larger than the sum of the remaining two, and (3) says that one *specified* variable from the triple is strictly larger than the sum of the remaining two.

Observe that (2) and (3) are mutually exclusive. For any nonnegative integer values of x_{e_1}, x_{e_2}, and x_{e_3}, (2) is valid if and only if (3) is not valid.

DEFINITION 2.1. *A* local linear program (LLP) *associated with a cubic graph* $G = (V, E)$ *is a system of linear constraints*

$$\begin{aligned} A_i x &\geq b_i \quad (i = 1, \ldots, n), \\ x &\geq 0, \end{aligned} \tag{4}$$

where each $A_i \geq b_i$ *is a block associated with vertex* $i = 1, \ldots, n$.

We can associate 4^n distinct local linear programs with a given cubic graph since there are three options in (3). Not every local linear program is feasible. Let us present two examples of infeasible local linear programs.

EXAMPLE 2.2. *Consider the following LLP associated with* K_4*:*

$$\begin{aligned} x_{12} &\geq x_{13} + x_{14} + 1 && \text{block of vertex } 1, \\ x_{23} &\geq x_{12} + x_{24} + 1 && \text{block of vertex } 2, \\ x_{13} &\geq x_{23} + x_{34} + 1 && \text{block of vertex } 3, \\ \left.\begin{aligned} x_{14} &\leq x_{24} + x_{34} \\ x_{24} &\leq x_{14} + x_{34} \\ x_{34} &\leq x_{14} + x_{24} \end{aligned}\right\} && \text{block of vertex } 4, \end{aligned}$$

$$x_{12}, x_{13}, x_{14}, x_{23}, x_{24}, x_{34} \geq 0.$$

This LLP is infeasible, since summing the first three inequalities gives $x_{14}+x_{24}+x_{34} \leq -3$, *which contradicts the nonnegativity constraint.*

It is convenient to depict a given LLP directly on the underlying graph. This is done so that an edge $e = ij$ is oriented from i to j whenever the block of the vertex j is the constraint (3) saying that x_e is larger than the sum of the other two variables. In this way we obtain a graph with some oriented and some unoriented edges, which fully describes a given LLP. The orientation of K_4 corresponding to the LLP of Example 2.2 is shown in Fig. 1.

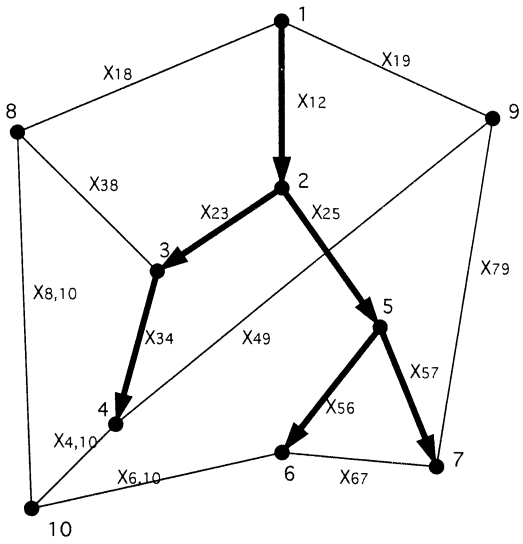


FIG. 2. *Graph* G_{10}.

EXAMPLE 2.3. *Consider the LLP given in Fig.* 2 *by the graph* G_{10} *and the partial orientation of its edges. This LLP is infeasible, which will be shown later in Example* 4.6.

An important property of local linear programs is that a program is feasible if and only if it has an integer solution where every variable is bounded by $2n-1$. This result will be proved in §4. As a by-product, we will obtain a combinatorial characterization of the feasibility of local linear programs (cf. condition (γ) in Theorem 4.5). For technical reasons, it is convenient to introduce a notion of a *generic* vector.

DEFINITION 2.4. *A vector* $z = (z_e)$ *is called* generic *if* (i) z *is strictly positive, i.e.,* $z_e > 0$ *for every* e, *and* (ii) $z_e \neq z_f + z_g$ *for every triple of edges* e, f, g *incident to a common vertex.*

THEOREM 2.5. *Let* $G = (V, E)$ *be a cubic graph together with an associated local linear program* (4). *The following are equivalent:*

(α) *The LLP is feasible.*

(β) *LLP has a generic integer feasible solution* $x = (x_e)$, *which is bounded such that* $1 \leq x_e \leq 2n-1$ *for every* e.

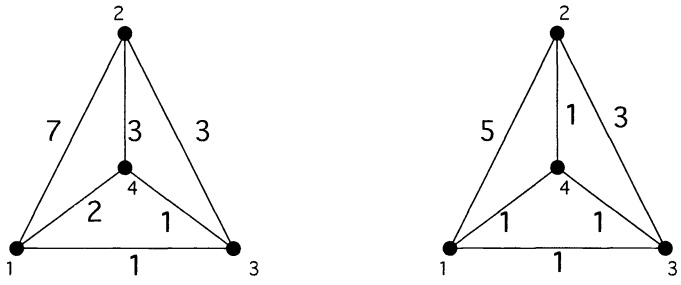
The proof will be given in §4.

3. FLIP local search in cubic graphs. Let $G = (V, E)$ be a cubic graph and $c \in Z^E$ be a vector of integer edge weights. Any subset $S \subset V$ determines a 2-partition $(S, V \setminus S)$ of vertices and a *cut* $\delta(S) := \{ij \in E \mid i \in S, j \notin S\}$. The *weight* of a cut $\delta(S)$ is defined as

$$c(\delta(S)) := \sum_{e \in \delta(S)} c_e. \tag{5}$$

The *max-cut* problem asks one to find a subset S for which $c(\delta(S))$ is maximum. The max-cut problem is NP-complete even when restricted to cubic graphs with c identically 1 for all edges (see [11]). The FLIP local search (defined below) is the simplest local search heuristic for max-cut, and some more involved heuristics (e.g., the Kernighan–Lin neighborhood) may bring better quality of local optima.

We say that a 2-partition $(T, V \setminus T)$ is in the *FLIP neighborhood* of $(S, V \setminus S)$ if $|S \Delta T| = 1$, where $S \Delta T$ denotes the symmetric difference of sets S and T. In other words, the FLIP



(a) *nongeneric c.* (b) *generic w.*

FIG 3

neighborhood consists of the 2-partitions obtained by moving a single vertex of S or $V \setminus S$ into the opposite class. A sequence of 2-partitions

$$(S_0, V \setminus S_0), \ldots, (S_N, V \setminus S_N) \tag{6}$$

is called a *FLIP local search* if $(S_i, V \setminus S_i)$ is in the FLIP neighborhood of $(S_{i-1}, V \setminus S_{i-1})$ and $c(\delta(s_i)) > c(\delta(s_{i-1}))$ for every $i = 1, \ldots, n$. Integer N is called *the number of steps* or the *length* of the local search (6). Observe that we do not require that $(S_N, V \setminus S_N)$ be a local optimum.

Since the FLIP neighborhood is the only neighborhood structure used in the paper, we will omit the term "FLIP" in the rest of the paper. Hence, *local search* and *local max-cut* will always mean the local search and the local maximum cut with respect to the FLIP neighborhood, respectively. On the other hand, we will often need to distinguish the local search with respect to different edge weight functions. In that case we will write *c-local search* to denote the local search with respect to c. Furthermore, we will adopt the following terminology: A vertex i is said to be *c-happy* with respect to S if moving i to the opposite partition class of $(S, V \setminus S)$ does not increase the weight of the cut. That is, i is c-happy if and only if

$$c(\delta(S \Delta i)) \leq c(\delta(S)). \tag{7}$$

Since a 2-partition $(S, V \setminus S)$ is fully determined by one partition class, we will write a c-local search as a sequence $\mathcal{S} = (S_0, S_1, \ldots, S_N)$, where $S_{t+1} \Delta S_t = \{i_t\}$ and vertex i_t is not c-happy in $\delta(S_t)$, $t = 0, \ldots, N-1$. Let us illutrate the local search in the following example.

EXAMPLE 3.1. *Consider K_4 with the edge weights c given in Fig.* 3(a). *The sequence* $(\{3\}, \{3,4\}, \{2,3,4\}, \{2,3\}, \{2\})$ *is a FLIP c-local search since*

$$c(\delta\{3\}) = 5 < c(\delta\{3,4\}) = 7 < c(\delta\{2,3,4\}) = 10 < c(\delta\{2,3\}) = 12 < c(\delta\{2\}) = 13.$$

The last set in the sequence is a c-local max-cut. In case we consider the edge weights w given in Fig. 3(b), *the local search sequence can be extended by one more member, i.e.,* $(\{3\}, \{3,4\}, \{2,3,4\}, \{2,3\}, \{2\}, \{2,4\})$ *is a FLIP w-local search since the corresponding sequence of the weights $w(\delta(S))$ is* $(5, 6, 7, 8, 9, 10)$.

Our main result is the following theorem.

THEOREM 3.2. *Let G be a cubic graph with an edge-weight function c.*

(i) *Then any FLIP local search has length at most $6n^2$.*

(ii) *If the edge-weights c are nonnegative, then it has length at most $3n^2$.*

Before proving Theorem 3.2 we need to prepare some auxiliary lemmas. It is convenient to prove the result first for cubic graphs with *nonnegetive* integer edge weights. Given a vector $c = (c_e) \in Z_+^E$ of nonnegative integer edge weights, we define the *local linear program determined by* c, denoted by LLP(c), as the system of linear inequalities consisting of the nonnegativity constraint

$$x \geq 0$$

and a block of the following constraints for every vertex i (where e, f, and g is the triple of the incident edges):

$$\begin{array}{lll} x_f + x_g \geq x_e & & c_f + c_g \geq c_e, \\ x_e + x_g \geq x_f & \text{if and only if} & c_e + c_g \geq c_f, \\ x_e + x_f \geq x_g & & c_e + c_f \geq c_g, \end{array} \tag{8}$$

or

$$x_e \geq x_f + x_g + 1 \quad \text{if and only if} \quad c_e \geq c_f + c_g + 1, \tag{9}$$

where e, f, and g denote the triple of the edges incident to a vertex i. The local linear program LLP(c) has the following trivial properties.

LEMMA 3.3. (i) *The vector c is a feasible solution of LLP(c).*

(ii) *A nonnegative integer vector $w = (w_e)$ is a feasible solution of LLP(c) if and only if LLP(c) = LLP(w).*

Proof. Part (i) follows immediately from the definition since the inequalities are defined so that each of them is satisfied by c. If LLP(c)=LLP(w), then w is a feasible solution of LLP(c) by part (i). Conversely, if w is not feasible for LLP(c), then there exists a vertex i such that, say, w satisfies (8) and c satisfies (9). Then w and c define different local linear programs, which differ in the block associated with vertex i. This proves part (ii). □

The feasible solutions of an LLP(c) do not uniquely determine the structure of locally maximum cuts, because if w and c have LLP(w)=LLP(c), they may still have distinct local maxima. As an example, consider K_4 with edge weights c and w given in Figs. 3(a) and 3(b). Then LLP(c)=LLP(w) but the local max-cuts are not the same. That is, the 2-partition $(\{2\}, \{1, 3, 4\})$ is a local max-cut with respect to c but not with respect to w. However, such a situation cannot occur when we restrict ourselves to the *generic* feasible solutions. We have the following lemma.

LEMMA 3.4. *Let w be a generic feasible solution of LLP(c). Then every w-happy vertex is c-happy with respect to any subset $S \subset V$.*

Proof. Let S be a given subset of vertices. Assume that a vertex i is not c-happy. We have to show that i is not w-happy either. Let e, f, and g be the edges incident to i. We have to distinguish two cases.

(i) Assume $c_e \geq c_f + c_g$. Then $e \notin \delta(S)$, since otherwise i would be c-happy. Since w is feasible and generic for LLP(c), we have $w_e \geq w_f + w_g + 1$. Hence vertex i is not w-happy in S either.

(ii) Assume $c_e < c_f + c_g$. Then $|\delta(S) \cap \{e, f, g\}| \leq 1$, since otherwise i would be c-happy. Since w satisfies

$$\begin{array}{r} w_e + w_f - w_g \geq 1, \\ w_e - w_f + w_g \geq 1, \\ -w_e + w_f + w_g \geq 1, \\ w \geq 1, \end{array} \tag{10}$$

i is not w-happy in S either. □

An immediate consequence of Lemma 3.4 is the following corollary.

COROLLARY 3.5. *If $\mathcal{S} = (S_0, S_1, \ldots, S_N)$ is a c-local search and w is a generic feasible solution of LLP(c), then the sequence $\mathcal{S}$ is a c-local search as well.*

Proof. By the definition of the local search, $v_i = S_i \Delta S_{i-1}$ is not c-happy for any $i = 1, \ldots, n$. By the above lemma, v_i is not w-happy either. Hence the sequence $\mathcal{S}$ is a w-local search as well. □

Let us remark that the converse of Corollary 3.5 is not true. A counterexample was presented in Example 3.1. Now we are able to present the proof of our main result in case the edge weights are nonnegative.

Proof of Theorem 3.2 (ii). Given a cubic graph G and nonnegative integer edge weights $c = (c_e)$, Theorem 2.5 guarantees that LLP(c) has a generic integer solution w, which is bounded so that $1 \leq w_e \leq 2n - 1$ for every e. Since G has $\frac{3}{2}n$ edges, any local search in (G, w) can take at most $3n^2$ steps, because every flipping increases the weight of a current cut by at least one, and the total weight of all edges is bounded by $3n^2$. Because of Corollary 3.5, the same bound also applies to any local search in (G, c), because any local search with respect to c is a local search with respect to w. □

Now, we extend the result to cubic graphs with arbitrary, i.e., not necessarily nonnegative, weights. We will make use of an operation of switching (see below), and the upper bound $3n^2$ will be replaced by $6n^2$ for the general case. From now on, the integer edge weights $c = (c_e)$ are arbitrary, i.e., possibly negative.

Let $S \Delta S'$ denote the symmetric difference of sets S and S'. Given $c \in Z^m$ and $S \subset V$, the *S-switching* of c is the vector $c^S = (c_e^S)$ defined by changing the sign of the weight on the edges in the cut $\delta(S)$, i.e.,

$$c_e^S := \begin{cases} -c_e & \text{for } e \in \delta(S), \\ c_e & \text{otherwise.} \end{cases} \tag{11}$$

Switching has been introduced by several authors for different purposes. We will employ switching to modify a given c to c^S such that a *specified* vertex i becomes incident only to *nonnegative* edges in c^S.

Let us mention that, in general, a weighted cubic graph *cannot* be reduced to nonnegative weights by switching. The "best" we can achieve by switching is that every vertex is incident to at most one negative edge. Switching a vertex (more precisely, the one element set consisting of a single vertex) which is incident to two or three negative edges reduces the total number of negative edges of the cubic graph.

The following lemma is well known and easy to check.

LEMMA 3.6. *Let $S, S' \subset V$, and c^S be the switching of a weight function c. Then*

(i) [1] $c^S(\delta(S' \Delta S)) = c(\delta(S')) - c(\delta(S))$;

(ii) *a vertex i is c-happy in $\delta(S')$ if and only if it is c^S-happy in $\delta(S' \Delta S)$;*

(iii) *$\delta(S')$ is a local maximum with respect to c if and only if $\delta(S' \Delta S)$ is a local maximum with respect to c^S.*

Proof. Part (i) is proved, e.g., in [1] and can easily be checked. Part (ii) is derived from (i) as follows: Assume that a vertex v is c^S-happy in $S' \Delta S$. Then, by the definition

$$c^S(\delta(S' \Delta S \Delta v)) \leq c^S(\delta(S' \Delta S)) \tag{12}$$

and part (i), we have

$$c^S(\delta(S' \Delta S \Delta v)) = c^S(\delta(S' \Delta v)) - c(\delta(S)).$$

Hence $c(\delta(S')) \geq c(\delta(S'\Delta v))$ is equivalent to (12). Part (iii) is an immediate consequence of (ii). □

Proof of Theorem 3.2 (i). Given $c \in Z^m$, let $\overline{c} = (\overline{c}_e) \in Z^m$ be defined by $\overline{c}_e = |c_e|$ for every $e \in E$. Consider the LLP($\overline{c}$), the local linear program determined by $\overline{c}$. Since LLP($\overline{c}$) is feasible ($\overline{c}$ is a solution), it also has a bounded generic solution $\overline{w}$ by part (β) of Theorem 2.5. Let us define $w = (w_e)$ by

$$w_e := \begin{cases} -\overline{w}_e & \text{if } c_e < 0, \\ \overline{w}_e & \text{if } c_e \geq 0. \end{cases} \tag{13}$$

Let $\mathcal{S} = (S_0, S_1, \ldots, S_N)$ be a *local search* with respect to w. We claim that $N \leq 6n^2$. Since $|w_e| \leq 2n - 1$ for every e, and $\frac{3}{2}n(2n-1) < 3n^2$, we have $-3n^2 < w(\delta(S)) < 3n^2$. Since $w(\delta(S_{t+1})) > w(\delta(S_t))$ for every t, N is bounded by $6n^2$. This proves the claim.

It remains to show that any local search $\mathcal{S}$ with respect to c is a local search with respect to w. In other words, we have to show that if a vertex i is not c-happy, then it is not w-happy either. This is easy to see when c, and thus also w, are nonnegative. In order to simplify the proof for a general c, we employ the operation of switching.

Assume that a vertex i is not c-happy in S_t. Let e, f, and g be the edges incident to i. Let c^S be a switching of c with respect to the set $S = \{j \mid c_{ij} < 0\}$. Then all c^S_e, c^S_f, c^S_g are nonnegative since the switching changes the sign of the weight on the edges in the cut $\delta(S)$. Since $\overline{c}_e = c^S_e, \overline{c}_f = c^S_f, \overline{c}_g = c^S_g$ (we recall that the bar means the absolute value for the entries of c), w^S_e, w^S_f, w^S_g are also nonnegative by the definition of w. Since i is not c-happy in S_t, i is not c^S-happy in $S\Delta S_t$. Using Lemma 3.4 with c^S, w^S and $S\Delta S_t$ instead of c, w, and S, respectively, we conclude that i is not w^S-happy in $S\Delta S_t$. Using Lemma 3.6 again, i is not w-happy in S_t. Hence, any c-local search is a w-local search as well. Since the number of steps for the w-local search is bounded by $6n^2$, the same bound is valid as the upper bound on the length of a c-local search. □

Remark. There are instances of weighted cubic graphs where the local search for a local maximum cut may take $\Theta(n^2)$. This shows that the bound of Theorem 3.2 is asymptotically optimal.

EXAMPLE 3.7. *Let P_n be a path on n vertices with edge weights c where $c_{i,i+1} = i$, $i = 1, 2, \ldots, n-1$. We show that there is a local search on P_n which requires $\binom{n}{2}$ steps. This result presents a lower bound on the worst case for cubic graphs as well. For n even, the path P_n can be embedded into a cubic graph, and the weights of the additional edges are set as zero.*

Proof. Assume that n is even. (The proof for n odd is quite analogous, hence we omit it.) We will describe a FLIP local search sequence with the following properties:

(i) The initial set of the sequence is $\{1, 2, \ldots, n\}$.

(ii) The final set of the sequence is $\{2, 4, 6, \ldots, n\}$.

(iii) At each step, the weight of the cut is increased precisely by 1.

It immediately follows from properties (i), (ii), and (iii) that the length of the constructed sequence is $\binom{n}{2}$, since the weight of the initial cut is 0 and the weight of the final cut $c(\delta(\{1, 2, \ldots, n\}))$ is $\sum_{i=1}^{n-1} i = \binom{n}{2}$. Observe that the final cut is the max-cut. The local search sequence is generated by the following algorithm:

$S := \{1, 2, \ldots, n\}$ (initialization)
For $j := n$ step -1 to 2 do
For $i := 1$ to $j - 1$ do
$S := S\Delta\{i\}$

We claim that the above algorithm has the properties (i), (ii), and (iii). In order to analyze the correctness of the algorithm, let us define sets S_{ij}, $1 \le i < j \le n$, as follows:

$$S_{ij} := \begin{cases} \{i, i+1, \ldots, j-1, j\} \cup \{j+2, j+4, , \ldots, n-2, n\} & \text{for } j \text{ even}, \\ \{1, 2, \ldots, i-1\} \cup \{j+1, j+3, \ldots, n-2, n\} & \text{for } j \text{ odd}. \end{cases}$$

Now, the following is easy to check:

(i)	$S_{1n} = \{1, 2, \ldots, n\}$	(the initial set),
(ii)	$S_{ij}\Delta\{i\} = S_{i+1,j}$	if $i < j-1$,
(iii)	$S_{ij}\Delta\{i\} = S_{1,j-1}$	if $i = j-1$,
(iv)	$S_{12}\Delta\{1\} = \{2, 4, 6, \ldots, n\}$	(the final set).

The weight of the cut increases by 1 both in steps (ii) and (iii). □

4. A combinatorial characterization of feasibility. Our goal is to provide a combinatorial criterion of solvability of local linear programs (4). We have to introduce several notions in order to formulate the combinatorial condition (γ) in Theorem 4.5, which extends the result formulated in Theorem 2.5. (A reader may follow all these notions in Fig. 2 and Example 4.6.) Assume that a local linear program associated with a cubic graph $G = (V, E)$ is given. Graph G and LLP are considered fixed. We will classify the vertices and edges with respect to a given LLP. For a vertex i, let e, f, and g denote the triple of the edges incident to i. Let us say that an edge e (or f or g, respectively) *dominates* at i if $x_e \ge x_f + x_g + 1$ is the block of the given LLP.

Classification of vertices. If one of the incident edges dominates at a vertex i, the vertex is said to be *dominated*. If none of the incident edges dominates at i, the vertex is called *undominated*. The set of dominated and undominated vertices is denoted by D and U, respectively.

Classification of edges. An edge $e = ij$ which dominates at one or both of its end vertices is called *dominating*. The set of edges dominating at one end vertex is denoted by $\vec{E}$. The set of edges dominating at both end vertices is denoted by $\overleftrightarrow{E}$. The nondominating edges are called *plain* and are denoted by $\overline{E}$. Thus, we have $E = \overline{E} \cup \vec{E} \cup \overleftrightarrow{E}$. Let us remark that a local linear program associated with a cubic graph $G = (V, E)$ is fully determined by the decomposition $E = \overline{E} \cup \vec{E} \cup \overleftrightarrow{E}$.

The edges from $\overleftrightarrow{E}$ are called *bidirected*. The edges from $\vec{E}$ are called *directed* and are oriented towards the vertex where they dominate. The plain edges $\overline{E}$ are further classified as inner, outer, and border. The set of *inner* plain edges is defined as $\hat{E} = \{ij \in \overline{E} \mid i, j \in U\}$. The set of *border* plain edges is defined as $\{ij \in \overline{E} \mid i \in U, j \in D\}$. Finally, the set of *outer* plain edges is defined as $\{ij \in \overline{E} \mid i, j \in D\}$. (We do not introduce any symbols for the latter two sets.)

Auxiliary subgraphs of G. We introduce several auxiliary subgraphs of $G = (V, E)$. The set $\vec{E}$ of dominating edges spans a digraph $\vec{G} = (\vec{V}, \vec{E})$. (The set $\overleftrightarrow{E}$ of bidirected edges does not belong to $\vec{G}$.) Observe that the indegree of every vertex of $\vec{G}$ is at most one, since at most one edge can dominate at a vertex. The vertices of $\vec{G}$ of indegree zero are called *roots*.

The set of plain edges $\overline{E}$ spans an undirected graph $\overline{G} = (\overline{V}, \overline{E})$. The set of inner plain edges $\hat{E}$ spans an undirected graph $\hat{G} = (U, \hat{E})$.

A *mixed path* is a path $P = (v_1, \ldots, v_s)$ such that each pair (v_i, v_{i+1}) is either a directed edge of $\vec{E}$ or an undirected edge of $\overline{E}$, where the directed edges can be used only in the direction of P. (Bidirected edges are not used in a mixed path.)

The following lemma formulates the first necessary condition on the feasibility of (4).

LEMMA 4.1. *If $\vec{E}$ contains a directed cycle, then the LLP* (4) *is infeasible.* (*Bidirected edges are not considered as cycles.*)

Proof. Assume that $\vec{E}$ contains a directed cycle consisting of edges $e_1, \ldots, e_k$, $k \geq 3$. Then, by the definition of set $\vec{E}$, together with $x \geq 0$, we have

$$x_{e_t} \geq x_{e_{t+1}} + 1 \quad (t = 1, \ldots, k-1),$$
$$x_{e_k} \geq x_{e_1} + 1,$$

which proves that (4) is infeasible. □

As an example, consider the LLP given in Fig. 1 and discussed in Example 2.2. We will call $\vec{E}$ *acyclic* if it does not contain any cycle. A rooted tree which is directed out of the root is called a *branching*. The following lemma is straightforward because the indegree of every vertex of $\vec{E}$ is at most one.

LEMMA 4.2. *Assume that $\vec{E}$ is acyclic. Then*

(i) *every connected component of $\vec{E}$ is a branching;*

(ii) *every root of $\vec{G}$ is root of a branching.*

From now, we will assume that $\vec{E}$ is acyclic. Hence every component T of $\vec{G}$ is a branching. Each branching T has a uniquely determined *root*, which is denoted by $r(T)$. The outdegree of a root can be one, two, or three; the directed edges leaving $r(T)$ are called *root edges*.

Let us denote by $K(T)$ the connected component of the graph $\hat{G}$ which contains the root $r(T)$. $K(T)$ is called the *root component* of T. (Recall that the root $r(T)$ is undominated and $K(T)$ consists of undirected edges only.) Let $T_1, \ldots, T_p$ be the connected components of $\vec{E}$. We will classify the components $T_1, \ldots, T_p$ of $\vec{E}$ as *saturated* or *unsaturated*. The saturated components are either *basic saturated*, which are defined as those satisfying one of the conditions (s1)–(s4) below, or reachable in the sense of (s5) from a basic saturated component.

DEFINITION 4.3. *A component T of $\vec{E}$ is called* basic saturated, *if at least one of the following conditions* (s1)–(s4) *is satisfied:*

(s1) *The root $r(T)$ of T is incident to a bidirected edge.*

(s2) *The outdegree of the root $r(T)$ of T is greater than one.*

(s3) *The root component $K(T)$ contains, besides $r(T)$, a root $r(T')$ of a distinct branching T'.*

(s4) *The outdegree of the root $r(T)$ is one and the root component $K(T)$ of T contains a cycle.*

DEFINITION 4.4. *A component T of $\vec{E}$ is called* saturated, *if either* (i) *T is basic saturated or* (ii) *condition* (s5) *holds, where*

(s5) *there exists a mixed path $P = (v_1, \ldots, v_s)$ terminating in the root $v_s = r(T)$ of T such that the initial vertex v_1 is either* (i) *a root $r(T') = v_1$ of a basic saturated component T', or* (ii) *incident to a bidirected edge.*

Examples of branchings satisfying (s1)–(s5) are given in Figs. 4–8 below. Now we can present an extension of Theorem 2.5.

THEOREM 4.5. *Let $G = (V, E)$ be a cubic graph together with an associated local linear program* (4). *The following are equivalent:*

(α) *LLP* (4) *is feasible;*

(β) *LLP* (4) *has a generic integer feasible solution $x = (x_e)$, which is bounded such that $1 \leq x_e \leq 2n - 1$ for every e;*

(γ) *digraph $\vec{G}$ is acyclic, and every component T of $\vec{G}$ is saturated.*

Proof. The implication (β) $\Rightarrow$ (α) is trivial, since every generic feasible solution is feasible.

Part $(\alpha) \Rightarrow (\gamma)$. Assume that (4) is feasible. Then $\vec{G}$ is acyclic by Lemma 4.1. For a contradiction, assume that not all the components of $\vec{G}$ are saturated. Let $T_1, \ldots, T_\ell$ be the list of all unsaturated components of $\vec{G}$. We recall that every component T_t has a unique vertex called a root and a unique root edge (since the outdegree of a root of an unsaturated component is one). Let $K_1 := K(T_1), \ldots, K_\ell := K(T_\ell)$ denote the root components of $T_1, \ldots, T_\ell$ defined above. Since the unsaturated components do not meet (s4), we have that

$$\text{every } K_t,\ t = 1, \ldots, \ell, \text{ is a tree.} \tag{14}$$

Let $B_t, t = 1, \ldots, \ell$, denote the subset of border plain edges incident to K_t. Since T_t meets neither (s1) nor (s2), we have that

$$\text{the degree of } r(T_t) \text{ in } K \cup B_t \text{ is two, } t = 1, \ldots, \ell. \tag{15}$$

On the other hand, let F_t denote the border plain edges which are incident to $T_t, t = 1, \ldots, \ell$. Set $F := \bigcup_{t=1}^{\ell} F_t$ and $B := \bigcup_{t=1}^{\ell} B_t$. We claim that

$$B \subset F. \tag{16}$$

Claim (16) follows from the fact that the set of border edges cannot be incident to any saturated component. If it were, it would provide a mixed path by means of which some of T_t is hung on a saturated component, contradicting our assumption that all T_t are unsaturated.

Let a_t denote the root edge of an unsaturated component $r(T_t)$. We claim that

$$x_{a_t} \leq \sum_{e \in B_t} x_e. \tag{17}$$

By (15), there are two plain edges, say f and g, incident to $r(T_t)$; since $r(T_t)$ is an undominated vertex, the inequality $x_{a_t} \leq x_f + x_g$ should be valid. Then (17) follows by repeated application of inequalities (2) to the root $r(T_t)$ and all its descendants in the tree K_t.

On the other hand, repeatedly applying (3) to the vertices of the tree T_t, we obtain

$$x_{a_t} > \sum_{e \in F_t} x_e \tag{18}$$

for every $t = 1, \ldots, \ell$. Now (18) and (17) together give

$$x(F) = \sum_{e \in F} x_e < \sum_{t=1}^{\ell} x_{a_t} \leq \sum_{t=1}^{\ell} \sum_{e \in B_t} x_e = \sum_{e \in B} x_e = x(B). \tag{19}$$

Hence $x(F) < x(B)$, which contradicts $B \subset F$ by (16). This concludes the proof of $(\alpha) \Rightarrow (\gamma)$.

EXAMPLE 4.6. *We will illustrate our notions and the proof of infeasibility on the LLP associated with the graph* G_{10} *given in Fig.* 2. *We have*

$D = \{2, 3, 4, 5, 6, 7\}$ — *the set of dominated vertices,*
$U = \{1, 8, 9, 10\}$ — *the set of undominated vertices,*
$\vec{E} = \{12, 23, 34, 25, 56, 57\}$ — *directed edges,*
$\overleftrightarrow{E} = \emptyset$ — *bidirected edges,*
$\overline{E} = \{(1, 8), (1, 9), (3, 8), (4, 9), (4, 10), (6, 7), (6, 10), (7, 9), (8, 10)\}$ — *plain edges,*
$\hat{E} = \{(1, 8), (1, 9), (8, 10)\}$ — *inner plain edges,*
$\{(3, 8), (4, 9), (4, 10), (6, 10), (7, 9)\}$ — *border plain edges.*

The digraph $\vec{G} = (\vec{V}, \vec{E})$ has only one component T with the root $r(T) = 1$. The root component $K(T)$ is identical with $\hat{E}$. Since $K(T)$ is a tree and T has only one root edge, branching T is not saturated. Hence the LLP is infeasible by Theorem 4.5. *Let us apply the proof to our example. Consider the following inequalities involving the dominating edges:*

$$\begin{aligned} x_{1,2} &> x_{2,3} + x_{2,5}, \\ x_{2,3} &> x_{3,4} + x_{3,8}, \\ x_{3,4} &> x_{4,9} + x_{4,10}, \\ x_{2,5} &> x_{5,6} + x_{5,7}, \\ x_{5,6} &> x_{6,7} + x_{6,10}, \\ x_{5,7} &> x_{6,7} + x_{7,9}. \end{aligned}$$

As a consequence of the above system and the nonnegativity, we have

$$x_{1,2} > x_{4,8} + x_{4,9} + x_{4,10} + x_{6,10} + x_{7,9}. \tag{20}$$

On the other hand, consider the following system of inequalities associated with the plain vertices:

$$\begin{aligned} x_{1,2} &\leq x_{1,8} + x_{1,9}, \\ x_{1,8} &\leq x_{3,8} + x_{8,10}, \\ x_{8,10} &\leq x_{4,10} + x_{6,10}, \\ x_{1,9} &\leq x_{4,9} + x_{7,9}. \end{aligned}$$

As a consequence of the above system and the nonnegativity, we have

$$x_{1,2} \leq x_{4,8} + x_{4,9} + x_{4,10} + x_{6,10} + x_{7,9}, \tag{21}$$

which contradicts (20). *This proves that the given LLP associated with G_{10} is infeasible.*

Part $(\gamma) \Rightarrow (\beta)$. Let us assume that $\vec{G}$ is acyclic and all its components are saturated. Let us say that an integer vector x satisfying $1 \leq x \leq 2n - 1$ is *feasible* for a vertex i if $A_i x \geq 1$. Clearly, an integer vector x is generic and feasible for the given LLP if and only if it is feasible for every vertex i. We will construct a feasible generic vector x in two stages. In stage one we define an integer vector x which is feasible for all vertices but the roots. In stage two x will be iteratively augmented until it is feasible for all vertices. Let us note that during the whole process of creating it, the constructed x will have the property that, for every vertex i of $\overline{V}$, $x_e = x_f \geq x_g$ for the edges incident to i.

Stage 1. Set

$$x_{ij} := \begin{cases} 1 & \text{for } ij \in \overline{E} \quad \text{(undirected edge)}, \\ 2\nu(i, j) - 1 & \text{for } (i, j) \in \vec{E} \quad \text{(directed edge)}, \\ 2M & \text{for } (i, j) = (r_1, r_2) \quad \text{(bidirected edge)}, \end{cases} \tag{22}$$

where $M = \max(\nu(r_1, r_2), \nu(r_2, r_1)) - 1$ and $\nu(i, j)$ is the number of vertices in the subtree of $\vec{G}$ consisting of the edge (i, j) and all descendants of j in $\vec{G}$. (If $(i, j) = (r_1, r_2)$ is a bidirected edge, then the value of x_{r_1,r_2} is defined as the maximum of the two values which are obtained for the branching, consisting of directed edge (r_1, r_2) and all descendants of r_2, and directed edge (r_2, r_1) and all descendants of r_1. An example of the labeling by (22) is given in Fig. 4.

We claim that

$$\begin{aligned} &x \text{ defined by (22) is feasible for all vertices } i \\ &\text{distinct from roots of the components of } \vec{G}. \end{aligned} \tag{23}$$

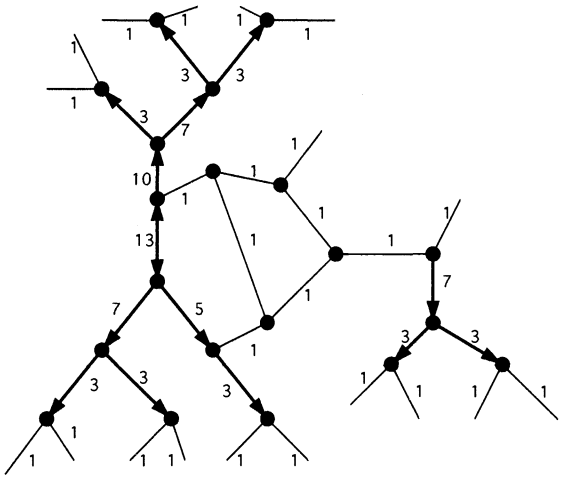


FIG. 4. *Labeling defined in Stage* 1.

Proof of the claim. We have to distinguish several cases. (i) If i is an undominated vertex which is not a root, then all three edges incident to it have weight 1 in x, and i is feasible.

(ii) If $e = ij$ is an edge dominating at a leaf of a branching, then $x_e = 2 \cdot 2 - 1 = 3$ by (22). Since the other two edges incident to leaf j have weight 1, we have $3 > 1 + 1$, and hence the leaf j is feasible in x.

(iii) Let i be a dominated vertex which is not a leaf and has an outdegree 2 in $\vec{G}$. Let e be the dominating edge entering i, and f and g be dominating edges leaving i. Since $\nu(e) = \nu(f) + \nu(g)$ we have

$$x_e = 2\nu(e) - 1 = (2\nu(f) - 1) + (2\nu(g) - 1) + 1 = x_f + x_g + 1,$$

and hence i is feasible in x.

(iv) Let i be as in case (iii) but with outdegree 1 in $\vec{G}$. Let e be the dominating edge entering i, f be the dominating edge leaving i, and G be the plain edge incident to i. Since $\nu(e) = \nu(f) + 1$ and $x_g = 1$, we have

$$x_e = 2\nu(e) - 1 = (2\nu(f) - 1) + x_g + 1 = x_f + x_g + 1,$$

and hence i is feasible in x.

(v) Let (r_1, r_2) be a bidirected edge. Let f_i, g_i denote a pair of the edges incident to r_i but not to r_j, where $i = 1, 2$ and $j = \{1, 2\} \setminus i$. Applying a proper case of (ii)–(iv), we have

$$x_{r_1 r_2} \geq 2\nu(r_j, r_i) \geq x_{f_i} + x_{g_i}$$

for $i = 1, 2$. Hence both r_1 and r_2 are feasible in x. This concludes the proof of the claim.

Stage 2. In order to construct a solution that is also feasible for the roots, we will repeatedly augment x on some edges. The augmentation will be defined so that the new x always remains feasible for all the vertices for which the old x was feasible. We distinguish several cases, depending on which of the conditions (s1)–(s5) saturates a component.

(i) In this step, x will be augmented to become feasible for the roots of components saturated by condition (s5).

The definition of the components saturated because of property (s5) can be alternatively formulated as follows: Let $\mathcal{S}$ initially denote the set of basic saturated components. Choose

a pair T, T' of components such that $T' \in \mathcal{S}$ and $T \notin \mathcal{S}$, and the mixed path P from the root $r(T')$ to the root $r(T)$ has the smallest possible number of edges in $\overline{G}$. If such a pair is selected, add T to the set $\mathcal{S}$ and say that T *hangs on* T' by P. Then repeat the process with another pair of components until all the components are in $\mathcal{S}$.

In order to augment x, take a component T which is not basic saturated, and such that no other component hangs on T. Let T' be the (unique) component on which T hangs and P be the mixed path from the $r(T')$ to $r(T)$ by which T hangs. Since P has the minimum possible number of edges in $\overline{G}$, it is easy to see that P consists precisely of one initial directed segment going through T' and an undirected segment connecting a vertex of T' to the root $r(T)$. Let ξ denote the current value of $x_{r(T),j}$, where $(r(T), j)$ is the root edge of $r(T)$. Let us augment x by

$$x_e := \begin{cases} x_e + \xi & \text{if } e \text{ is a directed edge of } P, \\ \max(x_e, \xi) & \text{if } e \text{ is an undirected edge of } P, \\ x_e & \text{if } e \text{ does not belong to } P. \end{cases} \tag{24}$$

It is easy to check that x becomes feasible for the root $r(T)$ and remains feasible for all the previous vertices. Repeat the augmentation (22) for other components that are not basic saturated. An example of updating by (24) is given in Fig. 5.

The only fact we have to check is that all the entries of x remain bounded by $2n - 1$. We prove this claim by the induction on the number p of the components of $\vec{G}$. If $p = 0$ then x is identically 1 by (22). If $p = 1$ then $\max_{e \in E} x_e = 2|T_1| - 1$ by (22), where $|T|$ denotes the number of vertices of a branching T. Now let $p > 2$. In order to avoid the technical details, consider the first application of (24) when a branching T is hung on a branching T'. Let e and e' denote the root edges of T and T', respectively. By (22), the old value of x_e and $x_{e'}$ is $2|T| - 1$ and $2|T'| - 1$, respectively. After augmenting x by (24) the edge e' becomes the heaviest edge of $T \cup T' \cup P$, and the new value of $x_{e'}$ is $2(|T| + |T'|) - 1$. This shows that the value of the heaviest edge remains bounded as required.

Before we present details of the augmenting procedure for the remaining cases (ii)–(iv), let us mention some features common to all three cases.

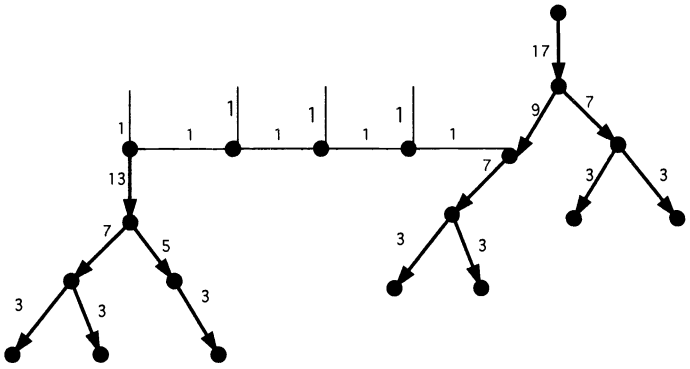
- Unlike case (i), the maximum value of the weight is not changed. This means that the upper bound $x_e \leq 2n - 1$ also remains valid after updating x.
- All undominated vertices i incident to some edges with updated weight have the following property after updating: if e, f, and g denote the triple of edges incident to i and $x_e \geq x_f \geq x_g$, then

$$x_e = x_f \geq x_g. \tag{25}$$

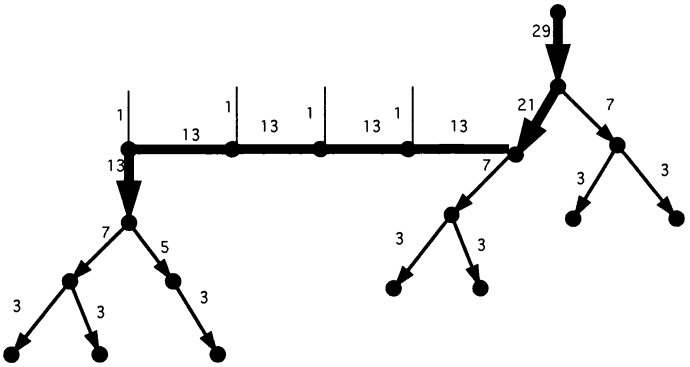
  An immediate consequence of (25) is that x satisfies the block (2), and hence x is feasible for i.
- If the value of x_e is increased on a root edge $e = ri$ and the value on the other two edges incident to i are retained, then x remains feasible for i, because inequality (3) is still valid.

(ii) In this step, x will be augmented to become feasible for the roots of components saturated by condition (s2). Let T be a component whose root $r(T)$ has outdegree 2 or 3. Let $e_i, i \in I$, denote the the root edges of T, where $|I| = 2$ or 3. Set $\xi := \max_{i \in I} x_{e_i}$, and augment x by

$$x_e := \begin{cases} \xi & \text{if } e \text{ is a root edge of } T, \\ x_e & \text{otherwise.} \end{cases} \tag{26}$$



(a) *Stage* 1 *labeling of a branching saturated by* (s5).



(b) *Stage* 2 *updating of a branching saturated by* (s5).

FIG. 5

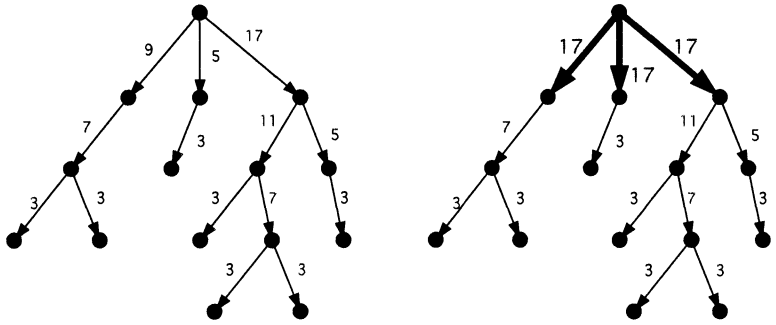
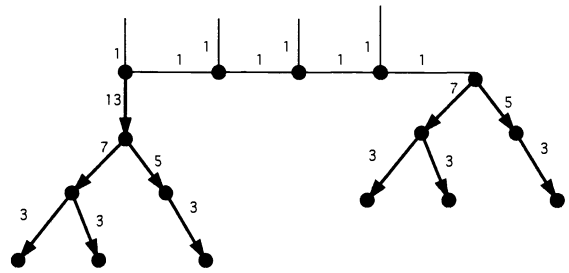


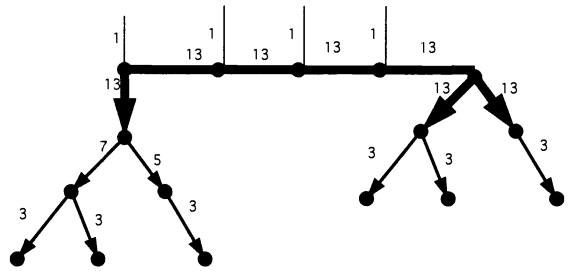
FIG. 6. *Stage* 1 *labeling and Stage* 2 *updating of a branching saturated by* (s2).

In other words, all the root edges receive the same (maximum) value, and x is unchanged elsewhere. It is easy to check that x becomes feasible for the root $r(T)$ and remains feasible for all the previous vertices. An example of updating by (26) is given in Fig. 6.

(iii) In this step, x will be augmented to become feasible for the roots of components saturated by condition (s3).



(a) *Stage* 1 *labeling of a branching saturated by* (s3).



(b) *Stage* 2 *updating of a branching saturated by* (s3).

FIG. 7

The condition (s3) defines a transitive relation on the pairs (T, T'). If both T, T' and T', T'' meet (s3) then T, T'' meet it as well. Hence the relation (s3) is an equivalence.

Let $T_1, T_2, \ldots, T_\ell$ be a maximal collection (i.e., an equivalence class) of components of $\vec{G}$ which pairwise meet (s3). It is easy to see that we can find a tree L which is a subgraph of $\overline{G}$ and such that the leaves of L are precisely the roots $r(T_1), \ldots, r(T_\ell)$. Let e_i denote the unique root edge of T_i and set $\xi := \max(x_{e_i} \mid i = 1, \ldots, \ell)$. Let us define the new x by

$$x_e := \begin{cases} \xi & \text{if } e = e_i \text{ for } i = 1, \ldots, \ell\,, \\ \max(x_e, \xi) & \text{if } e \in L, \\ x_e & \text{elsewhere.} \end{cases} \tag{27}$$

An example of updating by (27) is given in Fig. 7.

(iv) In this step, x will be augmented to become feasible for the roots of components saturated by condition (s4).

Let T be a basic saturated component resulting from (s4) but not (s3). Let $L = C \cup P$ be a subgraph of $\overline{G}$ which is a 1-sum of a cycle C and a path P, such that $r(T)$ is one end-vertex of P and the other is in C. (P may have zero length, in which case $L = C$ and $r(T)$ is a vertex of C.)

Let e_0 denote the unique root edge of T and set $\xi := x_{e_0}$. Let us define the new x by

$$x_e := \begin{cases} \xi & \text{for } e \in L, \\ x_e & \text{otherwise.} \end{cases} \tag{28}$$

□

An example of updating by (28) is given in Fig. 8.

Acknowledgments. The author thanks an anonymous referee for many helpful comments concerning the details of the presentation.

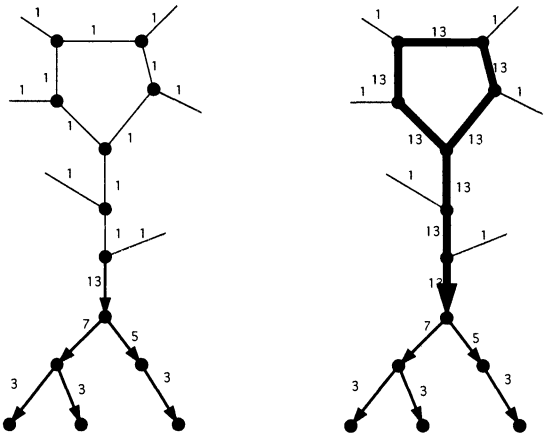


FIG. 8. *Stage* 1 *labeling and Stage* 2 *updating of a branching saturated by* (s4).

REFERENCES

[1] F. BARAHONA AND R. MAHJOUB, *On the cut polytope*, Math. Programming, 36 (1986), pp. 157–173.

[2] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability*, W. H. Freeman, San Francisco, CA, 1979.

[3] A. HAKEN AND M. LUBY, *Steepest descent can take exponential time for symmetric connection networks*, Complex Systems, 2 (1988), pp. 191–196.

[4] D. S. JOHNSON, C. H. PAPADIMITRIOU AND M. YANNAKAKIS, *How easy is local search?*, J. Comput. System Sci., 37 (1988), pp. 79–100.

[5] M. W. KRENTEL, *Structure in locally optimal solutions*, in Proc. 30th IEEE Symposium on Foundations of Computer Science, Washington, DC, IEEE Computer Society Press, 1989, pp. 216–221.

[6] ———, *On finding locally optimal solutions*, SIAM J. Comput., 19 (1990), pp. 742–749.

[7] M. LOEBL, *Efficient maximal cubic graph cuts*, Proc. 18th International Colloquim on Automata, Languages and Programming (ICALP), 1991, Lecture Notes in Computer Science 510, Springer-Verlag, Berlin, Heidelberg, pp. 351–362.

[8] C. H. PAPADIMITRIOU, *The complexity of the Lin–Kerninghan heuristic for the traveling salesman problem*, SIAM J. Comput., 21 (1992), pp. 450–465.

[9] S. POLJAK, *On existence theorems*, Discrete Math., 122 (1993), pp. 423–434.

[10] A. A. SCHÄFFER AND M. YANNAKAKIS, *Simple local search problems that are hard to solve*, SIAM J. Comput., 20 (1991), pp. 56–87.

[11] M. YANNAKAKIS, *Node- and edge-deletion NP-complete problems*, Proc. 10th ACM Symposium on Theory of Computing, ACM Press, New York, 1978, pp. 253–264.

SIAM J. COMPUT.
Vol. 24, No. 4, pp. 840–858, August 1995

010

EASILY CHECKED GENERALIZED SELF-REDUCIBILITY*

LANE A. HEMASPAANDRA† AND RICCARDO SILVESTRI‡

Abstract. This paper explores two generalizations within NP of self-reducibility: Arvind and Biswas's kernel constructibility and Khadilkar and Biswas's committability. Informally stated, kernel constructible sets have (generalized) self-reductions that are easy to check, though perhaps hard to compute, and committable sets are those sets for which the potential correctness of a partial proof of set membership can be checked via a query to the same set (that is, via a self-reduction). We study these two notions of generalized self-reducibility on nondense sets. We show that sparse kernel constructible sets are of low complexity, extend previous results showing that sparse committable sets are of low complexity, and provide structural evidence of interest in its own right—namely, that if all sparse disjunctively self-reducible sets are in P then FewP ∩ coFewP is not P-bi-immune—that our extension is unlikely to be extended further. We obtain density-based sufficient conditions for kernel-constructibility: sets whose complements are captured by nondense sets are perforce kernel constructible. Using sparse languages and Kolmogorov complexity theory as tools, we argue that kernel constructibility is orthogonal to standard notions of complexity.

Key words. self-reducibility, sparse sets, kernel constructibility, committable sets, ambiguity-bounded computation

AMS subject classifications. 68Q15, 03D15

1. Introduction.

Self-reducibility, the ability to convert the task of solving an instance of a problem to the task of solving (usually simpler) instances of the same problem, is a key to the modern understanding of complexity classes and their most important natural problems. Many of the most powerful results for NP can be viewed as by-products of the self-reducibility of the standard NP-complete problem SAT; for example, perhaps the most illuminating proof of the result of Karp and Lipton that "the polynomial hierarchy equals NP^{NP} if there exists a sparse NP-hard set" is the approach of using the base NP machine to guess the sparse set and the NP oracle to check that the guessed sparse set is consistent with SAT's self-reducibility tree both internally and at the leaves [25], [22]. Joseph and Young [23] have written an excellent survey on the importance of self-reducibility in computational complexity theory.

Motivated by the increasingly appreciated importance of self-reducibility and the widely acknowledged importance of the class NP, researchers have sought to understand the structure of self-reducibility within NP. Disjunctive self-reducibility, a type of self-reducibility possessed only by NP sets and possessed by SAT, is perhaps the most thoroughly investigated self-reducibility. However, in recent years researchers have studied many other reducibilities of varying levels of generality or restrictiveness (for example, the logspace self-reducibility of Balcázar, which led to results on space efficiency and sparse hard sets [20], and the strictly one word decreasing self-reducibility of Ogihara and Lozano [33], which also yields results about sparse hard sets [33]). The present paper is concerned with two particularly elegant generalizations of disjunctive self-reducibility: Arvind and Biswas's kernel constructibility [2], [3] and Khadilkar and Biswas's committability [26]. Each has the advantage of being possessed only by sets in NP—probably the most interesting domain on which to study self-reducibility—and of being provably at least as inclusive as disjunctive self-reducibility.

*Received by the editors July 27, 1992; accepted for publication (in revised form) April 8, 1994.

†Department of Computer Science, University of Rochester, Rochester, New York 14627 (lane@cs.rochester.edu). The work of this author was supported in part by National Science Foundation grants NSF-CCR-8957604, NSF-INT-9116781/JSPS-ENGR-207, and NSF-CCR-9322513, and was done in part while visiting the University of Electro-Communications, Tokyo, and the University of Ulm.

‡Department of Computer Science, University of Rome, 00198 Rome, Italy. The work of this author was supported in part by Ministero della Pubblica Istruzione through "Progetto 40%: Algoritmi, Modelli di Calcolo e Strutture Informative," and was done in part while visiting the University of Rochester.

Informally, a set is kernel constructible if it has an easily recognizable subset, its "kernel," to which every element of the set is connected via a short path each of whose edges can easily be checked.

DEFINITION 1.1 ([3], based on [2]). *We say that a set L is* kernel constructible *if there exist a set K, such that $K \subseteq L$ and $K \in$ P, a polynomial-time computable relation*[1] *R, and a polynomial q, such that*

1. *for every x and y, if xRy and x is in L then y is in L;*
2. *for every string $x \in L - K$, there exist a natural number ℓ and strings $y_0, \ldots, y_\ell$, such that*
   (a) $y_0 \in K$,
   (b) *for each* $1 \leq i \leq \ell$, $y_{i-1} R y_i$,
   (c) $y_\ell = x$, *and*
   (d) ℓ *and each* $|y_i|$ *are at most* $q(|x|)$.

We'll describe a path of the form above as a length ℓ path.

Following Arvind and Biswas, we'll refer to R as a constructing relation (for L) and K as a kernel (for L). KC denotes the class of all kernel constructible sets. In some sense, each application of R can be thought of as a nondeterministic self-reduction existing within the framework of a global constraint on the length of the self-reduction chain.[2]

It is immediate from the definitions that P $\subseteq$ KC $\subseteq$ NP and all disjunctively self-reducible sets are kernel constructible [2], [26]. Interestingly, kernel constructibility is also clearly a generalization, within NP, of paddability,[3] as first noted by Khadilkar and Biswas [26] and restated below in a slightly stronger form noted by Torán (personal communication).

PROPOSITION 1.2 (Torán). *Every paddable* NP *set is kernel constructible via a constructing relation that is symmetric and such that every nonkernel element of the set has a length one path.*

The proof of the above proposition simply makes the kernel consist of each instance of the problem padded with each solution of the instance, and makes the relation link such padded instances to the corresponding instances (and vice versa to obtain a symmetric relation). An example from Arvind and Biswas [2] that perhaps gives a better hint of the flavor of kernel constructibility considers the kernel consisting of graphs consisting of (only) a cycle among all the graph's nodes, and the relation "G_1 R G_2 if and only if G_2 is G_1 with a single edge added"; this kernel and constructing relation clearly yield the NP-complete problem of Hamiltonian cycles.[4]

However, even though all paddable sets are kernel constructible (and thus many NP-complete sets are kernel constructible as a result of the paddability results of Berman and Hartmanis [4]), many interesting sets are not paddable. For example, Joseph and Young constructed a specific artificial "k-creative" NP-complete set that has not yet been proven to be paddable (though many other k-creative sets such as the tiling problem of Homer are paddable [21]), and very recently it has been shown [27] that the decision version of the Reconstruction Conjecture (see [13])—considered by many to be the most important problem in applied graph theory in the wake of the resolution of the Four-Color Conjecture—is a seemingly

[1] That is, there is a polynomial-time computable set R, which we'll view as a binary relation: for $\langle \cdot, \cdot \rangle$, a standard pairing function as discussed in the proof of Theorem 2.3, we'll be interested in whether $\langle x, y \rangle \in R$ and, to emphasize that it is a relation, we'll sometimes write this as xRy.

[2] Note that self-reductions here need not always reduce strings to smaller elements. One can find in the literature other examples of self-reducibilities that don't always map to smaller strings, such as autoreducibility ([5]; see also [39]) and the self-reducibilities central in the study of polynomially enumerable sets [16].

[3] A set A is paddable [4] if it is a P-cylinder: $A \times \{0, 1, \ldots\}$ is polynomial-time isomorphic to A [32].

[4] In these two examples, we did not discuss the issue of the choice of the polynomial $q(\cdot)$ from the definition of kernel constructibility, as it is clear that choosing such a polynomial is easy.

nonpaddable set in NP that is provably Graph-Isomorphism-hard. Furthermore, sparse sets, sets with polynomially many elements at each length, are never paddable, and thus one cannot hope to prove such sets kernel constructible via paddability. In fact, §2.1 proves that sparse kernel constructible sets are quite simple: they can be accepted by a polynomial-time machine given a database for some set of low nondeterministic ambiguity (that is, they are in P^{FewP}). Section 2.1 also proves that any NP set whose complement is a subset of a simple nondense set is kernel constructible and establishes other sufficient conditions for kernel constructibility.

Section 2.2 explores the relationship between kernel constructibility and complexity classes based on nondeterministic machines. Although almost all familiar NP-complete sets are clearly kernel constructible ([2] via [4]), we show that such weak classes as UP $\cap$ coUP are not robustly (i.e., with respect to all oracles) kernel constructible and, in fact, cannot even be robustly approximated by kernel constructible sets: there is a relativized world in which UP $\cap$ coUP is KC-immune, and one can even witness this immunity via a sparse set. We also present results showing that co-sparse sets can also witness the weakness of kernel constructibility, and trivializing kernels does not suffice to trivialize kernel constructibility.

Section 3 discusses the related concept of committability [26], a notion of self-reducibility that attempts to characterize the class of problems Π for which one can reduce the question, "Is X a prefix of a solution to an instance π of Π?" to a question to Π; that is, one can check (via self-reduction) whether one has made a "good start" towards finding a proof. We link two seemingly different notions by proving that if all sparse disjunctively self-reducible sets are in P, then FewP $\cap$ coFewP is not P-bi-immune (i.e., for each FewP $\cap$ coFewP set, there is a P set that approximates well in the sense of finding an infinite subset of either the set or the set's complement). It follows from this that no extension of Khadilkar and Biswas's [26] Fortune-like [9] statement, "if a committable set A many-one reduces to a co-sparse set then A is in P," to the Mahaney-like [31] statement, "if a committable set A many-one reduces to a sparse set then A is in P," is possible, unless FewP $\cap$ coFewP is not P-bi-immune (an event we consider unlikely). We extend Khadilkar and Biswas's Fortune-like result right up to the edge of this limitation. Section 3 also establishes a result connecting disjunctive self-reducibility, committability, and kernel constructibility; although it is not known whether all committable sets are kernel constructible (after all, if P = NP, they all trivially are), it does, nonetheless, hold that, if there is a set that is committable but not disjunctively self-reducible, then there is a set that is committable and kernel constructible but not disjunctively self-reducible.

2. Density and kernel constructibility. Section 2.1 studies the complexity of sparse KC sets and provides a number of sufficient conditions for kernel constructibility. For example, all NP sets whose complements are subsets of any nondense P set are kernel constructible. Section 2.2 provides evidence that kernel constructibility is orthogonal to "class-based" notions of complexity: though some very hard sets are kernel constructible, some relatively simple classes are not robustly kernel constructible. Section 2.2 stresses separations via sparse languages and suggests the use of Kolmogorov complexity (as opposed to generalized Kolmogorov complexity) as a proof tool.

2.1. Sparse sets and sufficient conditions. The following definition makes the natural notion of relativized kernel constructibility explicit and allows one to speak of languages with kernels and constructing relations in classes beyond P. Note that it is identical to the definition (Definition 1.1 of this paper) of Arvind and Biswas, except that the kernel and the constructing relation are made members of potentially more general classes.

DEFINITION 2.1. *We say that a set L is in $(\mathcal{C}_1, \mathcal{C}_2)$-KC if there exist a set K such that $K \subseteq L$ and $K \in \mathcal{C}_1$, a $\mathcal{C}_2$-computable relation*[5] *R, and a polynomial q, such that*

1. *for every x and y, if xRy and x is in L then y is in L;*
2. *for every string $x \in L - K$, there exist a natural number ℓ and strings $y_0, \ldots, y_\ell$, such that*
   (a) *$y_0 \in K$,*
   (b) *for each $1 \leq i \leq \ell$, $y_{i-1}Ry_i$,*
   (c) *$y_\ell = x$, and*
   (d) *ℓ and each $|y_i|$ are at most $q(|x|)$.*

We'll describe a path of the form above as a length ℓ path.

Thus, a set is kernel constructible in the sense of Arvind and Biswas exactly if it is in (P, P)-KC, and thus, following their notation, we'll continue to denote (P, P)-KC by KC. Similarly, we'll use KC^A as a shorthand for $(\mathrm{P}^A, \mathrm{P}^A)$-KC. Various trivial statements immediately follow from the definition.

Observation 2.2.

1. For every pair of classes $\mathcal{C}$ and $\mathcal{D}$, $\mathcal{C} \subseteq (\mathcal{C}, \mathcal{D})$-KC.
2. If class $\mathcal{C}$ is closed downwards under many-one reductions, then $\mathcal{C} \subseteq (\{S| \, ||S|| \leq 1\}, \mathcal{C})$-KC.
3. NP = (P, NP)-KC = (NP, P)-KC = (NP, NP)-KC.

The following result says that sparseness, combined with kernel constructibility, puts severe limitations on the complexity of sets. Recall that a set L is $\mathrm{P}^{\mathcal{C}}$-printable if a deterministic oracle machine M and a set $A \in \mathcal{C}$ exist such that M^A, on input 0^n, prints all the elements of L of length at most n. (Note that, although all sparse NP sets are P^{NP}-printable, it is not known (and not likely) that all sparse NP sets are $\mathrm{P}^{\mathrm{FewP}}$-printable, because of the ambiguity of certifying membership.)

THEOREM 2.3. *If A is sparse and in* (FewP, FewP)-KC, *then A is $\mathrm{P}^{\mathrm{FewP}}$-printable.*

The proof of Theorem 2.3 first finds the kernel and then "grows" outward from the kernel. However, there are a number of perils that must be avoided. Of course, were we ever to binary search for census values, we might introduce great ambiguity, since $\binom{n}{(n/2)}$ is exponential in n. Also, we can't (in any obvious way) guess a proof that a given string is a member of our kernel constructible set, since there might be an exponential number of (short, legal) distinct paths from the kernel to that element, and thus the check would be an NP one rather than a FewP one. Even given a specified path, we can't (in any obvious way) check its validity in FewP, since every segment of the path can be certified in a polynomial number of ways and $(n^{\mathcal{O}(1)})^{(n^{\mathcal{O}(1)})}$ is not always $n^{\mathcal{O}(1)}$.

Though this is a more subtle point, we can't just iteratively feed the "distance j from the kernel" strings to our oracle and ask for prefixes of things that the constructing relation links to it; this is true because, although the fact that the set is sparse ensures that, for every x in the kernel constructible set, it holds that $C_x = \{y| \, \langle x, y\rangle \in R \wedge |y| \leq q(|x|)\}$ is of polynomially bounded cardinality, there is no reason to assume that, for x not in the kernel constructible set, it holds that C_x is even subexponential in size. One might say that, since we're growing iteratively, in the actual run of the algorithm we can indeed avoid ever passing an x outside of the set to the oracle; this is correct but does not help, as the FewP oracle must be FewP on every input, even on inputs that will never be asked on any actual run of the algorithm! (The class $\mathrm{P}^{\mathcal{FEWP}}$, obtained when one asks only for "Few-like" behavior only on queries actually obtained during an actual run of the algorithm, has been studied by Cai, Hemachandra, and

[5]That is, there is a set $R \in \mathcal{C}_2$, which we'll view as a binary relation: for $\langle \cdot, \cdot \rangle$ a standard pairing function as discussed in the proof of Theorem 2.3, we'll be interested in whether $\langle x, y\rangle \in R$, and, to emphasize that it is a relation, we'll sometimes write this as xRy.

Vyskoč [7] and motivated by work of Grollmann and Selman [12], and their evidence suggests that this class is very powerful indeed—so powerful that a $\mathrm{P}^{\mathrm{FewP}}$ result such as that of Theorem 2.3 is certainly much preferable to a $\mathrm{P}^{\mathcal{FEWP}}$ result.)

Proof of Theorem 2.3. Let $A \in$ (FewP, FewP)-KC $\cap$ SPARSE. Choose some K and R so that $K \in$ FewP is a kernel of A and $R \in$ FewP is a constructing relation for A built upon kernel K. Let $q(\cdot)$ be the related polynomial from the definition of kernel constructibility bounding the lengths of the "short paths of short strings" that lead to each string in A and the lengths of the strings along those paths. Let $\langle \cdot, \cdot \rangle_2$ be a 2-ary pairing function with the standard nice properties (a bijection between $\Sigma^* \times \Sigma^*$ and Σ^* that is polynomial-time computable and polynomial-time invertible, and is honest with respect to both arguments). Let $\langle \cdot, \ldots, \cdot \rangle_m$ be a multi-arity honest polynomial-time computable "pairing" function with the standard nice properties: although possibly not surjective, we can tell in polynomial time whether a given string is in the image of the function, and given $\langle x_1, \ldots, x_z \rangle_m$, we can compute z and $x_1, \ldots, x_z$ in time polynomial in $|\langle x_1, \ldots, x_z \rangle_m|$. For simplicity, we'll drop the 2 and m subscripts, since it will be clear from context which function we're referring to; places where there can *only* be 2-ary pairs will implicitly be $\langle \cdot, \cdot \rangle_2$, and other pairings will be $\langle \cdot, \ldots, \cdot \rangle_m$. Let Q be a polynomial-time binary predicate and k_Q be an integer such that $(\forall x, y)[(\langle x, y \rangle \in R \iff (\exists z)\, [|z| = |\langle x, y \rangle|^{k_Q} + k_Q \wedge \langle \langle x, y \rangle, z \rangle \in Q]) \wedge (\|\{z \mid |z| = |\langle x, y \rangle|^{k_Q} + k_Q \wedge \langle \langle x, y \rangle, z \rangle \in Q\}\| \leq |\langle x, y \rangle|^{k_Q} + k_Q)]$; since $R \in$ FewP, such Q and k_Q can be found (and one can, without loss of generality ensure, as implicitly assumed above, that the witness length bound and sparseness upper bound are the same).

We will now describe a $\mathrm{P}^{\mathrm{FewP}}$ procedure that, on input 0^M, will print all the elements of A of length at most M. First, our procedure will obtain $K^{\leq q(M)}$ via queries to the following FewP set: $L_1 = \{\langle 0^n, \text{pre}, \langle s_1, \ldots, s_z \rangle \rangle \mid$ there is a string $y \notin \{s_1, \ldots, s_z\}$ in $K^{\leq n}$ whose prefix is pre$\}$. Note that we do not use binary search, since that would cause too much ambiguity. (Of course, our prefix search via L_1 will discover long strings first, since short strings that are prefixes of longer strings will be obscured until the long string is made an s_i.)

For each $m \in N$, let us say that a string is in $\text{length}_m(0)$ if it is in $K^{\leq m}$; thus, we have already found $\text{length}_{q(M)}(0)$. For $i \geq 1$ and $m \in N$, let us say that a string is in $\text{length}_m(i)$ if it is not in $\bigcup_{\ell < i} \text{length}_m(\ell)$ and its membership in A is certified by a path of length[6] at most (and thus no less than, because of the earlier part of this definition) i, where the value of i and the lengths of all strings along the path are at most m.

Suppose we have already found $\text{length}_{q(M)}(j), \ldots, \text{length}_{q(M)}(0)$ and, for each string in any of these sets, we have found an appropriate type of certificate for the string's membership in its set. That is, for a string $\alpha_\ell \in \text{length}_{q(M)}(\ell)$, $\ell \leq j$, we will have found kernel string α_0, a string α_1 in $\text{length}_{q(M)}(1), \ldots,$ a string $\alpha_{\ell-1}$ in $\text{length}_{q(M)}(\ell - 1)$, such that $\langle \alpha_0, \alpha_1 \rangle \in R$, $\ldots, \langle \alpha_{\ell-2}, \alpha_{\ell-1} \rangle \in R$, and $\langle \alpha_{\ell-1}, \alpha_\ell \rangle \in R$, as well as certificates with respect to Q for each of the ℓ inclusions in R. We will now show how to extend this information to obtain the same information up to and including all of $\text{length}_{q(M)}(j+1)$.

For each string $w \in \text{length}_{q(M)}(j)$, do the following: Compute all strings z, $|z| \leq q(M)$, such that $\langle w, z \rangle \in R$ holds and $z \notin \bigcup_{0 \leq i \leq j} \text{length}_{q(M)}(i)$, via queries to the set L_2 below. (Note that from different strings $w \in \text{length}_{q(M)}(j)$, we may obtain the same string $z \in \text{length}_{q(M)}(j+1)$. After doing the following procedure for each w, we associate with each string $z \in \text{length}_{q(M)}(j+1)$ exactly one of the ways of reaching it; alternatively, in the queries to L_2 we could always include as s_i's all strings in $\text{length}_{q(M)}(j+1)$ that had been found for previous w's.)

$$L_2 = \{\langle 0^H, \langle y_0, \ldots, y_z \rangle, \text{pre}, \langle w_1, \ldots, w_z \rangle, \langle s_1, \ldots, s_{z'} \rangle, \text{prewit} \rangle \mid$$

[6] That is, the ℓ of Definition 2.1.

1. $y_0, \ldots, y_z, s_1, \ldots, s_{z'}$, and pre are each of length at most H, and
2. $y_0 \in K$, and
3. $(\forall i : 1 \leq i \leq z)\,[|\langle y_{i-1}, y_i\rangle|^{k_Q} + k_Q = |w_i| \wedge \langle\langle y_{i-1}, y_i\rangle, w_i\rangle \in Q]$, and
4. $(\exists v)\,[|v| \leq H \wedge$ pre is a prefix of $v \wedge v \notin \{s_1, \ldots, s_{z'}\} \wedge (\exists u)\,[|u| = |\langle y_z, v\rangle|^{k_Q} + k_Q \wedge$ prewit is a prefix of $u \wedge \langle\langle y_z, v\rangle, u\rangle \in Q]]$ }.

Note that $L_2 \in$ FewP. The search using this set is the natural one. That is, z will be j; y_z will be w; the y_i's will be the path we know to w; the w_i's will be our witnesses of the R relations along that path; the s_i's will be all strings in $\text{length}_{q(M)}(j+1)$ that the current w has yielded so far plus all strings in $\bigcup_{0 \leq i \leq j} \text{length}_{q(M)}(i)$; and prewit will remain ϵ while we find all of the $\text{length}_{q(M)}(j+1)$ strings that w generates.

Thus, using the above procedure, after cycling over each string $w \in \text{length}_{q(M)}(j)$, we have collected all of $\text{length}_{q(M)}(j+1)$. As noted above, let us associate with each string in $\text{length}_{q(M)}(j+1)$ a single w yielding it. Now we can use L_2 again, via manipulating prewit, to obtain witnesses with respect to membership in R connecting each $\text{length}_{q(M)}(j+1)$ string with the w we associated with it. (It is important to do this after $\text{length}_{q(M)}(j+1)$ is completely obtained, so that when searching for the witness of a string β, we can include as s_i's *all* the strings in $\bigcup_{0 \leq i \leq j+1} \text{length}_{q(M)}(j+1)$ except β.) Thus, using this procedure, in P^{FewP} we can find all the strings in $A^{\leq q(M)}$ whose membership in $A^{\leq q(M)}$ is certified by kernel elements, path elements, and path length, whose lengths, lengths, and value, respectively, are at most $q(M)$. So by the definition of kernel constructibility, this set intersected with $\Sigma^{\leq M}$ is exactly $A^{\leq M}$. □

COROLLARY 2.4. *If* $A \in \text{KC} \cap \text{SPARSE}$ *then* $A \in \text{P}^{\text{FewP}}$.

The following two theorems present density-based sufficient conditions for kernel constructibility. Theorem 2.6 uses Valiant's [38] "evaluation" and "checking" classes. Note that $\text{PC}_t^p \subseteq \text{PE}_t^p$ is a relatively strong assumption; clearly, $\text{PC}_t^p \subseteq \text{PE}_t^p \Rightarrow \text{P} = \text{NP} \cap \text{coNP}$ (which even holds in the stronger version in which the functions are not polynomially output length bounded [38, Prop. 5]).

DEFINITION 2.5 ([38]).

1. PE_t^p *is the class of total multivalued functions* f *for which there exist a polynomial-time function* g *and a polynomial* p *such that, for any* x, $g(x) \in f(x)$ *and for any* $y \in f(x)$, $|y| \leq p(|x|)$.
2. PC_t^p *is the class of total multivalued functions* f *such that the length of the outputs of* f *is bounded by a polynomial and the language* $\{\langle x, y\rangle \mid y \in f(x)\}$ *belongs to* P.

THEOREM 2.6. *Suppose that* $\text{PC}_t^p \subseteq \text{PE}_t^p$. *If* $L \in \text{NP}$ *is such that there is a* k *such that* $(\forall n)\,[||\overline{L} \cap \Sigma^n|| \leq 2^{n-n^{1/k}}]$, *then* L *is kernel constructible.*

Proof. Since $L \in \text{NP}$, a polynomial-time relation S and a nondecreasing polynomial p exist such that, for every x,

$$x \in L \iff (\exists w)\,[|w| = p(|x|) \wedge xSw].$$

Consider the multivalued function f defined for every $x \neq \epsilon$ by

$$f(x) = \{xyw \mid |y| = |x|^{2k} \wedge |w| = p(|xy|) \wedge xySw\}$$

and having $f(\epsilon) = \{\epsilon\}$. Since f is checkable in polynomial time, in order to show that $f \in \text{PC}_t^p$, it is enough to verify that f is total. By hypothesis, $||\overline{L} \cap \Sigma^{n+n^{2k}}|| \leq 2^{n+n^{2k}-(n+n^{2k})^{1/k}}$. It follows that $||\overline{L} \cap \Sigma^{n+n^{2k}}|| < 2^{n^{2k}}$ for all $n \geq 1$. This implies that, for every x with $|x| = n > 0$, there exists at least one string y such that $|y| = |x|^{2k}$ and $xy \in L$; thus a string w exists such that $xyw \in f(x)$.

Since $PC_t^p \subseteq PE_t^p$, a polynomial-time function g exists such that, for any x, $g(x) \in f(x)$. Let

$$K = \{z \mid z = xy \text{ with } |y| = |x|^{2k} \wedge g(x) = xyw \text{ for some } w\}.$$

If $\epsilon \in L$, put the empty string into K as well. Let R be the relation defined by, for every z and x,

$$zRx \iff (z \in K \wedge xSh(z)),$$

where h is the function defined as follows:

$$h(z) = \begin{cases} u & \text{if } z = uv \text{ with } |v| = |u|^{2k}, \\ 0 & \text{otherwise.} \end{cases}$$

We show that K and R witness that L is kernel constructible. Clearly, $K \in P$ and $K \subseteq L$. It is also clear that, for any z and x, zRx implies $x \in L$. Now, let $x \in L - K$. Let w be a string such that xSw. Consider the string $g(w) = wuv$ with $|u| = |w|^{2k}$ and $|v| = p(|wu|)$. It holds that $wu \in K$ and, since $h(wu) = w$, we have $xSh(wu)$. Hence, we have $wuRx$. □

THEOREM 2.7. *Suppose that $L \in NP$ and a set $E \in P$ exists such that $\overline{L} \subseteq E$ and, for some k, it holds that $(\forall n)\ [||E \cap \Sigma^n|| \leq 2^{n-n^{1/k}}]$; then L is kernel constructible.*

Observe that from Theorem 2.7, we have the following corollary.

DEFINITION 2.8 (natural generalization of the NP-capturability of [6]). *For any class C and any set A, we say that A is* C-capturable *if there is a sparse set $B \in C$ such that $A \subseteq B$.*

COROLLARY 2.9. *If a set $L \in NP$ is such that $\overline{L}$ is P-capturable, then L is kernel constructible.*

Finally, we have two non-density-based sufficient conditions for kernel constructibility. We omit the proofs of Theorems 2.7, 2.10, and 2.11; interested readers may find the proofs in [18].

THEOREM 2.10. *Let L be a language in NP. If there exists a function f such that $f(\Sigma^*) \subseteq L$, $f(\Sigma^*) \in P$, f is one-to-one, and f^{-1} is polynomial-time computable,*[7] *then L is kernel constructible.*

THEOREM 2.11. *If L_1 is kernel constructible and f is a polynomial-time computable, honest*[8] *function such that $f(L_2) = L_1$ and $f(\overline{L_2}) \subseteq \overline{L_1}$, then L_2 is kernel constructible.*

2.2. Relativization results: Location of KC. Of course, KC is a subset of NP. Fenner, Fortnow, and Kurtz [8] have recently constructed an oracle relative to which all NP-complete sets are isomorphic, and thus relative to which all NP-complete sets are kernel constructible. Looking in the other direction, we show, via Kolmogorov complexity, that many classes of relatively low complexity are not robustly kernel constructible and indeed cannot even be robustly approximated by kernel constructible sets.

Kolmogorov complexity can be a powerful tool in oracle constructions (see [29]) as first shown by Hartmanis [14] and as further explored, for example, by Gavaldà et al. [11]

[7] By this we mean that ($\exists$ a polynomial-time computable function g)$(\forall y \in f(\Sigma^*))\ [f(g(y)) = y]$. Note that this flexible definition allows g to output any junk it likes on inputs that are not in the range of f. In many cases, the difference between inverses of this form and inverses that detect when their input is not in the image is trivial; for example, it is here given our assumption that $f(\Sigma^*) \in P$. However, note that in the model of inverse that we are assuming, we cannot drop the $f(\Sigma^*) \in P$ assumption; the fact that we have not assumed that f is in P precludes the standard trick of taking a purported preimage and seeing if it maps back from where it came.

[8] We'll say that a (possibly non-one-to-one) function f is honest if there is a polynomial q such that for every x and y, it holds that $f(x) = y \Rightarrow q(|y|) \geq |x|$.

and Gavaldà [10]. Most commonly, generalized Kolmogorov complexity, that is, resource-bounded Kolmogorov complexity, is used in such constructions. However, we feel that standard (nonresource-bounded) Kolmogorov complexity provides at least as clean a tool for proving oracle results. The proof of Theorem 2.12 demonstrates the use of Kolmogorov complexity in such a context.

It perhaps is worth spending a moment considering the merits of generalized Kolmogorov complexity, rather than dismissing it offhand. The only argument we can think of that would favor generalized Kolmogorov complexity over standard Kolmogorov complexity is that the latter tends to yield nonrecursive oracles, while the former often yields recursive oracles. But are recursive oracles any more interesting than nonrecursive oracles? The desire for recursive oracles is probably driven by a hope that recursive oracles are more valid predictors of the real world than nonrecursive oracles. We know of no evidence supporting this hope, and note in passing that in each case we know of involving nonrelativizable results [36], [19], the oracle that differs from the real world is, or can easily be made, a *recursive* oracle. Thus, we fail to find any compelling evidence that recursive oracles are inherently more desirable than nonrecursive oracles. The extent to which *any* oracle results give insight into the structure of computation is open to debate in light of the rise of nonrelativizable results; however, the scope of nonrelativizable techniques is currently quite limited, and Allender has argued persuasively that oracle constructions still yield valuable information about problems in complexity theory ([1], see also [15]).

As first suggested by Hartmanis for the case of generalized Kolmogorov complexity, for standard Kolmogorov complexity one also finds that immunity[9] *constructions* come surprisingly easily. The oracle constructions in the proofs of Theorems 2.12 and 2.13 are extremely clean and simple, though the proofs of the correctness of the constructions are somewhat sensitive. One must be very careful in creating such constructions; not all Kolmogorov oracle arguments in the literature are devoid of problems. We note that our two Kolmogorov constructions contain no explicit diagonalizations; in contrast, Hartmanis's seminal paper [14] uses an explicit diagonalization, which somewhat taints its use of Kolmogorov complexity, in light of Hartmanis's tremendously perceptive insight that "Kolmogorov complexity is prepackaged diagonalization."

THEOREM 2.12. *There is a relativized world A in which* $\mathrm{UP}^A \cap \mathrm{coUP}^A$ *is* KC^A*-immune. Indeed, there is a sparse set in* $\mathrm{UP}^A \cap \mathrm{coUP}^A$ *witnessing the immunity.*

Of course, in the above relativized world $\mathrm{KC}^A \neq \mathrm{NP}^A$ and not all NP^A sets (or even all $\mathrm{UP}^A \cap \mathrm{coUP}^A$ sets) are disjunctively self-reducible. We also note that the proof's sparse $\mathrm{UP}^A \cap \mathrm{coUP}^A$ set that is KC^A-immune is in fact "almost" kernel constructible. For some string x_0, this set is $(\{x_0\}, \mathrm{UP}^A \cap \mathrm{coUP}^A)$-KC via a constructing relation R that is unambiguous: for each $x \in \Sigma^*$, it holds that $||\{y|\, xRy\}|| \leq 1$.

Proof of Theorem 2.12. Let $\mathrm{K}[f(n)]$ indicate all strings x such that there is a y, $|y| \leq f(|x|)$, such that $M_{\mathrm{univ}}(y)$ prints x and halts, where M_{univ} is a fixed machine universal for Kolmogorov complexity (see [29]). Note that in this proof we won't use relativized Kolmogorov complexity ($\mathrm{K}^A[\cdot]$), relative Kolmogorov complexity ($\mathrm{K}(x|\, z)$), or generalized (i.e., resource-bounded) Kolmogorov complexity; we will only use standard Kolmogorov complexity. Let

$$\alpha_1 = 2^{1+2^{2^2}}, \quad \alpha_2 = 2^{1+2^{2^{2^2}}}, \quad \alpha_3 = 2^{1+2^{2^{2^{2^2}}}}, \quad \text{etc.},$$

[9]For any class of sets $\mathcal{C}$, we say that a set B is $\mathcal{C}$-immune if B is infinite yet has no infinite subsets in $\mathcal{C}$; we say that a class $\mathcal{D}$ is $\mathcal{C}$-immune if some infinite set in $\mathcal{D}$ is $\mathcal{C}$-immune (see [35]). Intuitively, if $\mathcal{D}$ is $\mathcal{C}$-immune, $\mathcal{C}$ is so weak relative to $\mathcal{D}$ that not only does $\mathcal{C}$ fail to contain $\mathcal{D}$, but also $\mathcal{C}$ sets cannot even "approximate infinitely often from the inside" some $\mathcal{D}$ set.

and let $\widehat{L} = \{\alpha_1, \alpha_2, \ldots\}$. For each $\ell \in \widehat{L}$, put into A the lexicographically smallest length ℓ string in $\overline{\mathrm{K}[15n/16]}$. This completes the construction of the oracle A.

We now show that A has the desired behavior. We'll use size-of(H) to denote the number of bits in the representation of H (H will always be a machine or a finite set), where we assume some standard and reasonably succinct convention on sizes of encodings of machines and finite sets. Let $T = \{x \mid (\exists y)\ [|y| = |x| \wedge xy \in A]\}$. Note that $T \in \mathrm{UP}^A \cap \mathrm{coUP}^A$. It is not hard to see that for some string x_0, T is $(\{x_0\}, \mathrm{UP}^A \cap \mathrm{coUP}^A)$-KC via a constructing relation R that is unambiguous: for each $x \in \Sigma^*$, it holds that $||\{y \mid xRy\}|| \leq 1$, since one can just walk up the chain of strings in A with the first half of the smallest string in A being the kernel.

CLAIM 1. *T is P^A-immune.*

Proof of Claim 1. Suppose S is an infinite P^A subset of T. Let $S = L(M_i^A)$, where, without loss of generality, deterministic polynomial-time machine M_i runs in time at most $n^i + i$ regardless of its oracle. Let x be any string in S such that $|x| >>$ size-of(M_i) and $|x|^i + i < \alpha_{z+1}$, where $|x| = \alpha_z/2$. Note that $\Sigma^{|x|} \cap L(M_i^{A^{\leq \alpha_z}}) = \Sigma^{|x|} \cap L(M_i^A)$, since, because of the wide spacing of A's strings, M_i on length $|x|$ strings cannot possibly query any element of A of length greater than α_z. On the other hand, though we know that $||\Sigma^{|x|} \cap L(M_i^{A^{\leq \alpha_z}})|| = 1$, it is plausible, offhand, that $\Sigma^{|x|} \cap L(M_i^{A^{\leq \alpha_z - 1}})$ might have some other number of strings; thus, we cannot go straight to Case 2 below, but must first dispose of Case 1.

Case 1. There is a length $|x|$ string w so that at some point in the run of $M_i^{A^{\leq \alpha_z}}(w)$, M_i queries the (unique) length α_z string in A.

Note that in this case, we can describe the length α_z string in A as "the string that $M_i^{\{v_1, v_2, \ldots, v_{z-1}\}}(w)$ queries on step foo," where the v_i's are all the $z-1$ strings of $A^{\leq \alpha_z - 1}$ and foo is the first step on which $M_i^{A^{\leq \alpha_z}}(w)$ queries the length α_z string in A. Note that the size of this description is at most about size-of(M_i) + $\alpha_z/2$ + $\log((\alpha_z/2)^i + i)$ + size-of($A^{\leq \alpha_z - 1}$) + const, where the constant is globally fixed;[10] because of the wide spacing of A, this is less than $15\alpha_z/16$, contradicting the fact that the length α_z string of A was in $\overline{\mathrm{K}[15n/16]}$. Thus, Case 1 is excluded.

Case 2. For no length $|x|$ string w does $M_i^{A^{\leq \alpha_z}}(w)$ query the length α_z string in A.

In this case, clearly $\Sigma^{|x|} \cap L(M_i^{A^{\leq \alpha_z}}) = \Sigma^{|x|} \cap L(M_i^{A^{\leq \alpha_z - 1}})$. Thus, in this case x has a short name (one could make the name even shorter by writing the bits of z rather than the bits of α_z, but there is no need to): "the unique string of length $\alpha_z/2$ in $L(M_i^{\{v_1, v_2, \ldots, v_{z-1}\}})$." This name is of size at most about size-of($A^{\leq \alpha_z - 1}$) + size-of(M_i) + $\lceil \log(\alpha_z/2) \rceil$ + const, where the constant is globally fixed. This is much smaller than $|x| = \alpha_z/2$, and so it follows that v_z, the length α_z string in A, also has a short name because of the fact that its first half is x: "the string whose last half is u and whose first half is described by $foo2$," where u is the last half of v_z (thus it takes $|x|$ bits to write u inside the name) and $foo2$ is the short name for x (in the short name for v_z, we actually write in the entire name for x). Thus v_z is not in $\overline{\mathrm{K}[15n/16]}$, and so our supposition that T is not P^A-immune is contradicted. □

CLAIM 2. *T is KC^A-immune.*

Proof of Claim 2. Using Claim 1, we will now show that T has no infinite KC^A subset. Suppose, by way of contradiction, that $S \subseteq T$ is in KC^A and S is infinite. Let $K \in \mathrm{P}^A$ and $R \in \mathrm{P}^A$ be a purported kernel and a constructing relation proving that $S \in \mathrm{KC}^A$. (If the set constructed by K and R is not S then clearly these are not viable candidates; henceforth, we can assume that K and R do construct S, and thus are an infinite subset of T.) Let $R = L(M_i^A)$, where, without loss of generality, M_i runs in time at most $n^i + i$ regardless of its oracle, and

[10] Throughout this paper all logarithms are base 2.

let $q(\cdot)$ be the associated polynomial of the definition of KC. By Claim 1, K is a finite set. Let string $r \in S$ be some string such that (1) r is very large relative to size-of(K) and size-of(M_i), (2) r also is so large that for all strings w with $|w| \leq |r|$ it holds that $|\langle w, r\rangle|^i + i$ is much shorter than the shortest string in A whose length is greater than $2|r|$ (a string which, if $|r| = \alpha_z/2$, is of length α_{z+1}), and (3) $q(|r|) < \alpha_{z+1}/2$, where $|r| = \alpha_z/2$. By our assumptions, there must be an $m \geq 2$ and $q_1, q_2, \ldots, q_{m-1}$ such that

1. $m, |q_1|, |q_2|, \ldots, |q_{m-1}|$ are each at most $q(|r|)$,
2. $q_1 \in K$, and
3. $\langle q_1, q_2\rangle \in R, \ldots, \langle q_{m-2}, q_{m-1}\rangle \in R$, and $\langle q_{m-1}, r\rangle \in R$.

Without loss of generality, we may assume that $r \notin \{q_1, \ldots, q_{m-1}\}$ (otherwise, truncate the path at the first occurrence of r). Note that by our assumptions, the length $(\alpha_{z+1})/2$ string in T (and thus any string in S of length greater than $\alpha_z/2$) is much too long to be a q_i (since otherwise, either the fact that $|q_i| \leq q(|r|)$ is violated or the fact that $S \subseteq T$ is violated). Since $S \subseteq T$, we thus know that $|q_{m-1}|$ is much shorter (at least about exponentially shorter) than $|r|$.

The argument is now similar to that of Claim 1. If, for some y, $|y| = |r|$, it holds that $M_i^A(\langle q_{m-1}, y\rangle)$ queries the length α_z string in A, then much as in Case 1 of Claim 1, this gives the length α_z string in A a short name that contradicts its membership in $\overline{\mathrm{K}[15n/16]}$. On the other hand, if for no string $|y|$, $|y| = |r|$, does $M_i^A(\langle q_{m-1}, y\rangle)$ query the length α_z string in A, then, much as in Case 2 of Claim 1, we can also argue that the length α_z string in A is not in $\overline{\mathrm{K}[15n/16]}$, since its first half has the short name "the unique length $\alpha_z/2$ string 'w' such that $M_i^{\{v_1, v_2, \ldots, v_{z-1}\}}(\langle q_{m-1}, \text{'}w\text{'}\rangle)$ accepts," where the v_i's are the $z-1$ strings in $A^{\leq \alpha_{z-1}}$ and 'w' is *not* some bit-string but rather a variable name. □

Thus, the result is established.

The above result implies that there is a world in which some NP^A set has only finite KC^A subsets. The first part of the proof of Theorem 2.12 makes the kernel finite (via making NP P-immune in the oracle world). Would that have sufficed to prove Theorem 2.12? The following shows that even if one forces all kernels to be finite (via immunity), one still may have hard KC sets.

THEOREM 2.13. *There is an oracle A and a set $L \in \mathrm{UP}^A \cap \mathrm{coUP}^A$ such that L is infinite, L is P^A-immune, yet $L \in \mathrm{KC}^A$. Indeed, the language L can be chosen to be sparse.*

COROLLARY 2.14. *There is a relativized world A in which* $\mathrm{SPARSE} \cap \mathrm{KC}^A \not\subseteq \mathrm{P}^A$.

The proof of Theorem 2.13 uses standard Kolmogorov complexity. We'll carefully interweave the structure of L and A, via definitions that repeatedly truncate and concatenate strings from L and A, in order to add new strings to L. Before proving Theorem 2.13, we state an intuitively clear lemma saying that one can find sequences of Kolmogorov complex strings that have strong independence properties. This result, Lemma 2.16, is proved via an easy counting argument.

When we speak of relative Kolmogorov complexity, e.g., $\mathrm{K}(\alpha_{i+1} | \alpha_i, \alpha_{i-1}, \alpha_{i-2}, \ldots, \alpha_0)$,[11] to be truly rigorous we must specify exactly what our model is of how the strings $\alpha_i, \ldots, \alpha_0$ are fed to the universal machine. Almost any reasonable model will work equally well for the purposes of our proof, but for clarity we specify a particular definition. (Note below that $\langle \cdot, \ldots, \cdot \rangle_m$ is (easily) computable and invertible, so it doesn't do any particular damage (beyond a constant, a level of flexibility already lost in fixing a universal machine).)

[11] In what follows (in particular, in the proof of Theorem 2.13), we'll often include the first few terms of a series, instead of just the first and last. This is because in some cases a term will be skipped, and we'll want to make it clear when this is the case. By specifying a large number of terms, we do not mean to imply that the series need have that number of terms. For example, when we write $\mathrm{K}(\alpha_{i+1} | \alpha_i, \alpha_{i-1}, \alpha_{i-2}, \ldots, \alpha_0)$, it will be perfectly valid for i to equal 0, in which case the written expression should be taken as a shorthand for $\mathrm{K}(\alpha_1 | \alpha_0)$.

DEFINITION 2.15 (see, e.g., [28]). *For any $z \geq 1$ and any strings x, $x_1, \ldots, x_z$, define*

$$\mathrm{K}(x \mid x_1, \ldots, x_z) = \min\{n \mid (\exists y)\, [|y| = n \wedge M_{\mathrm{univ}}(\langle y, x_1, \ldots, x_z\rangle_m) \text{ prints } x \text{ and then halts}]\}.$$

Whenever we write $\mathrm{K}(\cdots \mid S)$, it will be the case that S is a finite set and $\mathrm{K}(\cdots \mid S)$ in this case will be a shorthand for $\mathrm{K}(\cdots \mid z_{||S||}, z_{||S||-1}, \ldots, z_1)$, where $z_1 <_{\mathrm{lex}} \cdots <_{\mathrm{lex}} z_{||S||}$ are the elements of S. $\mathrm{K}(\cdots \mid y, S)$ for y a string and S a finite set is, similarly, a shorthand for $\mathrm{K}(\cdots \mid y, z_{||S||}, z_{||S||-1}, \ldots, z_1)$, where $z_1 <_{\mathrm{lex}} \cdots <_{\mathrm{lex}} z_{||S||}$ are the elements of S.

LEMMA 2.16. *There is a sequence of strings $\alpha_0, \alpha_1, \ldots$ such that*

1. $(\forall i \geq 0)\, [|\alpha_i| = 3 \cdot 2^{10+i}]$,
2. $(\forall i \geq -1)\, [\mathrm{K}(\alpha_{i+1} \mid \{\alpha_a \mid i \geq a \geq 0\}) \geq \frac{99}{100}|\alpha_{i+1}|]$, *and*
3. $(\forall i \geq 1)\, [\mathrm{K}(\alpha_i \mid \{\alpha_a \mid a = i+1 \text{ or } i > a \geq 0\}) \geq \frac{9}{10}|\alpha_i|]$.

Proof. Inductively, suppose we've already chosen $\alpha_0, \ldots, \alpha_\ell$, satisfying the unquantified statement inside part 1 of the lemma for every $0 \leq i \leq \ell$, the unquantified statement inside part 2 of the lemma for every $-1 \leq i \leq \ell - 1$, and the unquantified statement inside part 3 of the lemma for every $0 \leq i \leq \ell - 1$. (The base case $\ell = 0$, that is, choosing an α_0 of length $3 \cdot 2^{10}$ whose Kolmogorov complexity relative to the empty set of helping strings is at least $\frac{99}{100} \cdot 3 \cdot 2^{10}$, is trivial.) As a shorthand define, for each $i \geq 0$, $n_i = 3 \cdot 2^{10+i}$. Let $v_0, \ldots, v_{m-1}$ be any m strings. For each length n_{m+1} string β_{m+1}, at most $2^{\lfloor (9/10)n_m \rfloor + 1} - 1$ length n_m strings w can satisfy $\mathrm{K}(w \mid \beta_{m+1}, \{v_a \mid m > a \geq 0\}) \leq \frac{9}{10}n_m$. So, among all length n_m strings, at most $(2^{\lfloor (9/10)n_m \rfloor + 1} - 1)\, 2^{n_{m+1}}/(2^{n_{m+1}}/2) = 2 \cdot (2^{\lfloor (9/10)n_m \rfloor + 1} - 1)$ strings w satisfy: "For at least $\frac{1}{2}$ of all length n_{m+1} strings β, it holds that $\mathrm{K}(w \mid \beta, \{v_a \mid m > a \geq 0\}) \leq \frac{9}{10}n_m$."

Note that we can easily build a fixed machine that, whenever given any m strings $v_0, \ldots, v_{m-1}$ with $|v_i| = n_i$, recursively enumerates all length n_m strings w such that for at least $\frac{1}{2} \cdot 2^{n_{m+1}}$ length strings β is $\mathrm{K}(w \mid \beta, \{v_a \mid m > a \geq 0\}) \leq \frac{9}{10}n_m$, and the position within this enumeration gives any such string w a short name relative to $\{v_a \mid m > a \geq 0\}$. In particular, any such w satisfies $\mathrm{K}(w \mid \{v_a \mid m > a \geq 0\}) \leq \log(2 \cdot (2^{\lfloor (9/10)n_m \rfloor + 1} - 1)) +$ low-order terms. Since, by our assumption, α_ℓ satisfies $\mathrm{K}(\alpha_\ell \mid \{\alpha_a \mid \ell > a \geq 0\}) \geq \frac{99}{100}n_\ell$, α_ℓ cannot be such a w (relative to $\{\alpha_a \mid \ell > a \geq 0\}$). Thus, we may conclude that, for at least half of all length $n_{\ell+1}$ strings β, it holds that

$$\mathrm{K}(\alpha_\ell \mid \beta, \{\alpha_a \mid \ell > a \geq 0\}) \geq \frac{9}{10}n_\ell. \tag{1}$$

Among the at least $\frac{1}{2} \cdot 2^{n_{\ell+1}}$ strings β satisfying equation (1), note that for at most $2^{(99/100)n_{\ell+1}}$ it can hold that $\mathrm{K}(\beta \mid \{\alpha_a \mid \ell \geq a \geq 0\}) \leq \frac{99}{100}n_{\ell+1}$. Since $(\frac{1}{2} \cdot 2^{n_{\ell+1}}) - 2^{\frac{99}{100}n_{\ell+1}} \geq 1$, there is some β that, taken as $\alpha_{\ell+1}$, satisfies $|\alpha_{\ell+1}| = n_{\ell+1}$, $\mathrm{K}(\alpha_{\ell+1} \mid \{\alpha_a \mid \ell \geq a \geq 0\}) \geq \frac{99}{100}n_{\ell+1}$, and $\mathrm{K}(\alpha_\ell \mid \alpha_{\ell+1}, \{\alpha_a \mid \ell > a \geq 0\}) \geq \frac{9}{10}n_\ell$. □

Proof of Theorem 2.13.

Start of oracle construction.
Let $A = \{\alpha_0, \alpha_1, \ldots\}$, where the α_i are the strings of Lemma 2.16.
End of oracle construction.

The reader can see very clearly here the truth of Hartmanis's claim that Kolmogorov complexity is prepackaged diagonalization! We have only to verify that the A constructed above satisfies the requirements of the theorem. Let L be the set constructed from the finite kernel $\{1^{2^{11}}\}$ via the clearly P^A-computable relation defined by

$$aRb \iff (\exists i,\ p_1,\ p_2,\ q_1,\ q_2)\,[|a| = 2^{10+i} \wedge |b| = 2^{10+i+1} \wedge |p_1| = \tfrac{1}{4}|b| \wedge |p_2| = \tfrac{3}{4}|b| \wedge p_1 p_2 = b \wedge p_2 = \alpha_{i-1} \wedge |q_1| = |q_2| = |a|/2 \wedge q_1 q_2 = a \wedge q_2 = p_1].$$

That is, the ℓth string of L is made by concatenating the last half of the $(\ell - 1)$st string of L and the string $\alpha_{\ell-2}$. Note that, for every $i \geq 1$, L will have exactly one string at length 2^{10+i}; L will have no strings at any other lengths. Thus, we now know that L is an infinite, sparse set that is in KC^A. Furthermore, L is in $\mathrm{UP}^A \cap \mathrm{coUP}^A$, since clearly we can build an unambiguous [38] strong [30] machine for L that guesses (and checks via A) all the α_i of size at most the input size, and by the construction of L, the length 2^{10+i} string in L for $i \geq 3$ is the last two thirds of the bits of α_{i-3} followed by α_{i-2}.

It remains only to show that L is P^A-immune. Suppose, by way of contradiction, that there is a set S such that S is infinite, $S \in \mathrm{P}^A$, and $S \subseteq L$. Let M_i, without loss of generality running in time $n^i + i$ for every oracle, satisfy $L(M_i^A) = S$. Choose some string $x \in S$ such that $|x| \geq 1000, |x|^i + i << 2^{(1/100)|x|}$, and i+size-of(M_i)+ const_1+ const_2+ $\mathrm{const}_3 < \log\log|x|$, where the three constants are independent of x and M_i and will be implicitly defined below. Since $x \in S \subseteq L$, it holds that $|x| = 2^{10+j}$ for some j.

Case 1. There is a length 2^{10+j} string w for which $M_i^A(w)$ at some point in its computation queries a member of $\{\alpha_{j-1},\ \alpha_j,\ \alpha_{j+1}, \ldots\}$.

Let α_z be the member of $\{\alpha_{j-1},\ \alpha_j,\ \alpha_{j+1}, \ldots\}$ touched earliest in the run of $M_i^A(w)$ and suppose α_z is first queried on step s. Then, clearly,

"the string that $M_i^{\{\alpha_0,\ldots,\alpha_{j-2}\}}(w)$ queries on step s"

(note that we actually write into this name the actual values of each variable mentioned) is a short name proving that

$$\mathrm{K}(\alpha_z |\, \alpha_{j-2}, \ldots, \alpha_0) \leq \text{size-of}(M_i) + |w| + \log(|w|^i + i) + \mathrm{const}_1.$$

It follows from our assumptions above that

$$\begin{aligned} \mathrm{K}(\alpha_z |\, \alpha_{j-2}, \ldots, \alpha_0) &\leq \text{size-of}(M_i) + \frac{2}{3}|\alpha_z| + \log(2^{\frac{1}{100}|w|}) + \mathrm{const}_1 \\ &\leq \log\log\left(\frac{2}{3}|\alpha_z|\right) + \frac{2}{3}|\alpha_z| + \frac{1}{100}|w| \\ &\leq \log\log\left(\frac{2}{3}|\alpha_z|\right) + \frac{2}{3}|\alpha_z| + \frac{1}{150}|\alpha_z|. \end{aligned}$$

However, noting that $z - 1 \geq j - 2$, this contradicts the fact that $\mathrm{K}(\alpha_z |\, \alpha_{z-1}, \ldots, \alpha_0) \geq \frac{99}{100}|\alpha_z|$. Thus, Case 1 is excluded.

Case 2. Case 1 does not hold and, for some length 2^{10+j-2} string γ, it holds that $M_i^A(\gamma \cdot \alpha_{j-2})$ queries α_{j-3} at some point in its computation.

Note that $M_i^{A^{\leq|\alpha_{j-2}|}}(\gamma \cdot \alpha_{j-2})$ must also query α_{j-3}, since the failure of Case 1 means that longer "alpha" strings can't be touched here. Let s be the first step on which α_{j-3} is queried. In this case α_{j-3} has the name

"the string that $M_i^{\{\alpha_0,\ldots,\alpha_{j-2}\}}(\gamma \cdot \alpha_{j-2})$ queries on step s."

Note that this name proves that

$$\mathrm{K}(\alpha_{j-3} |\, \alpha_{j-2},\ \alpha_{j-4},\ \alpha_{j-5}, \ldots, \alpha_0) \leq |\gamma| + \text{size-of}(M_i) + \log((2^{10+j})^i + i) + \mathrm{const}_2.$$

Thus, from our assumptions it follows that

$$\begin{aligned}
&\mathrm{K}(\alpha_{j-3}|\,\alpha_{j-2},\,\alpha_{j-4},\,\alpha_{j-5},\ldots,\alpha_0)\\
&\leq \frac{2}{3}|\alpha_{j-3}| + \text{size-of}(M_i) + (1 + i(10 + j)) + \text{const}_2\\
&\leq \frac{2}{3}|\alpha_{j-3}| + \log\log\left(\frac{2^3}{3}|\alpha_{j-3}|\right) + \left(1 + \left(\log\log\left(\frac{2^3}{3}|\alpha_{j-3}|\right)\right)\left(3 + \log\left(\frac{|\alpha_{j-3}|}{3}\right)\right)\right).
\end{aligned}$$

This contradicts the fact that $\mathrm{K}(\alpha_{j-3}|\,\alpha_{j-2},\,\alpha_{j-4},\,\alpha_{j-5},\ldots,\alpha_0) \geq \frac{9}{10}|\alpha_{j-3}|$. Thus Case 2 is excluded.

Case 3. Case 1 does not hold and, for no length 2^{10+j-2} string γ, does $M_i^A(\gamma \cdot \alpha_{j-2})$ query α_{j-3} at some point in its computation.

Thus, for no length 2^{10+j-2} string γ does $M_i^A(\gamma \cdot \alpha_{j-2})$ query α_{j-3}, α_{j-1}, or $\alpha_j, \ldots$. This gives the last two thirds of α_{j-3} the short name

> "the unique string 'γ' of length two raised to the $10 + j - 2$ for which $M_i^{\{\alpha_0,\ldots,\alpha_{j-5},\alpha_{j-4},\alpha_{j-2}\}}(\text{`}\gamma\text{'}\cdot\alpha_{j-2})$ accepts."

This implies that

$$\mathrm{K}(\alpha_{j-3}|\,\alpha_{j-2},\,\alpha_{j-4},\,\alpha_{j-5},\ldots,\alpha_0) \leq \frac{1}{3}|\alpha_{j-3}| + \text{size-of}(M_i) + \log(10 + j - 2) + \text{const}_3.$$

It follows that

$$\begin{aligned}
&\mathrm{K}(\alpha_{j-3}|\,\alpha_{j-2},\,\alpha_{j-4},\,\alpha_{j-5},\ldots,\alpha_0)\\
&\leq \frac{1}{3}|\alpha_{j-3}| + \log\log(2^{10+j}) + \log(10 + j - 2)\\
&\leq \frac{1}{3}|\alpha_{j-3}| + \log\log(2^{3+\log(\frac{|\alpha_{j-3}|}{3})}) + \log\left(\left(3 + \log\left(\frac{|\alpha_{j-3}|}{3}\right)\right) - 2\right).
\end{aligned}$$

This contradicts the fact that $\mathrm{K}(\alpha_{j-3}|\,\alpha_{j-2},\,\alpha_{j-4},\,\alpha_{j-5},\ldots,\alpha_0) \geq \frac{9}{10}|\alpha_{j-3}|$. Thus, Case 3 is excluded.

Thus our assumption that L has an infinite P^A subset is contradicted. Thus L is indeed P^A-immune. □

The next result shows the existence of a relativized world in which the condition $E \in \mathrm{P}$ in Theorem 2.7 is necessary. Also, just as Theorem 2.12 showed that sparse sets can escape kernel constructibility, this theorem shows that co-sparse NP sets need not be kernel constructible (note that showing a world where NP has co-sparse P-immune sets does not suffice, since the possibility of infinite KC sets with finite kernels must also be eliminated); thus, from the following, we see that the claim "all co-sparse NP sets are kernel constructible" does not hold robustly.

THEOREM 2.17. *There exists an oracle H and a set $L \in \mathrm{NP}^H$ such that L is co-sparse and $L \notin \mathrm{KC}^H$.*

Proof. Let $M_1, M_2, \ldots$ be a standard enumeration of deterministic polynomial-time oracle Turing machines. We assume that M_i^X runs in time p_i, regardless of the oracle X, where $p_i(n) = n^i + i$. We can define an enumeration of polynomial-time oracle relations in the following way: for each i and oracle X, let R_i^X denote the relation defined by $(\forall x,\ y)\,[x R_i^X y \iff M_i^X(\langle x, y\rangle) \text{ accepts }]$.

Our goal is to construct an oracle H so that the language $L^H = \{x|\ (\exists y)\ [|y| = 8|x| \text{ and } xy \in H]\}$ is co-sparse and $L^H \notin \mathrm{KC}^H$. In order to guarantee that $L^H \notin \mathrm{KC}^H$, oracle H will

be such that L^H is P^H-immune (so that any kernel for L^H must be a finite set) and, for any i, either strings x, y exist such that $x \in L^H$, $y \notin L^H$, and $x R_i^H y$ or, for any sufficiently long string z and any $w \in L^H$ with $|w| < |z|$, $w R_i^H z$ does not hold. Thus, in either case, relation R_i^H cannot witness that $L^H \in \mathrm{KC}^H$.

We need the following notation. For each index i and oracle X, we say that a string x is an *M_i^X-killer* if $x \notin L^X$ and $M_i^X(x)$ accepts. Furthermore, we say that x is a *first M_i^X-killer* if x is an M_i^X-killer and, for any string y such that $|y| = |x|$ and $y < x$, y is not an M_i^X-killer. Similarly, we say that x is an *R_i^X-killer* if $x \notin L^X$ and there exists a z such that $|z| < |x|$, $z \in L^X$, and $z R_i^X x$. Finally, we say that x is a *first R_i^X-killer* if x is an R_i^X-killer and, for any string y such that $|y| = |x|$ and $y < x$, y is not an R_i^X-killer.

The oracle H will be constructed in stages. For each $n \geq 0$, let H_n represent the set of strings in H at the end of the nth stage of the construction. At each stage n, we will diagonalize against all the machines M_i and relations R_i with $i \leq \log n$, adding to H_{n-1} all the strings z for which there exist an x and a y so that (1) $z = xy$, $|x| = n$, and $|y| = 8n$, (2) x is neither a first M_i^X-killer nor a first R_i^X-killer, and (3) z does not belong to the set of strings queried during all the computations that have been considered in the previous stages and present stage. In order to specify this set of queries we introduce the following notation. For each index i, oracle X, and string x, let $Q(M_i^X(x))$ denote the set of strings queried during computation $M_i^X(x)$. For each n and oracle X, let

$$Q_n(X) = \bigcup_{i \leq \log n,\, |z| \leq n} \left(Q(M_i^X(z)) \cup \bigcup_{|w| \leq n} Q(M_i^X(\langle w, z \rangle)) \right)$$

The set of queries in question is equal to $Q_n(H_{n-1})$.

Stage 0. $H_0 := \emptyset$.

Stage $n > 0$. Consider the set I_n defined by $I_n = \{x \mid |x| = n \wedge (\exists i : i \leq \log n)$ [either x is a first $M_i^{H_{n-1}}$-killer or x is a first $R_i^{H_{n-1}}$-killer]$\}$. We then set $H_n := H_{n-1} \cup \{xy \mid |x| = n$ and $|y| = 8n$ and $x \notin I_n$ and $xy \notin Q_n(H_{n-1})\}$.

Clearly, $L^H \in \mathrm{NP}^H$.

The above construction is shown to be correct by the following claims.

CLAIM 3. *L^H is a co-sparse set.*

Proof of Claim 3. First of all, we show that, for each n, $||Q_n(H_{n-1})|| \leq 2^{7n}$. In fact, it holds that

$$\begin{aligned}
||Q_n(H_{n-1})|| &\leq \sum_{i \leq \log n,\, |z| \leq n} \left(||Q(M_i^{H_{n-1}}(z))|| + \sum_{|w| \leq n} ||Q(M_i^{H_{n-1}}(\langle w, z \rangle))|| \right) \\
&\leq \sum_{i \leq \log n} \sum_{k \leq n} \sum_{|z| = k} \left(||Q(M_i^{H_{n-1}}(z))|| + \sum_{h \leq n} \sum_{|w| = h} ||Q(M_i^{H_{n-1}}(\langle w, z \rangle))|| \right) \\
&\leq \sum_{i \leq \log n} \sum_{k \leq n} \left(2^k p_i(k) + \sum_{h \leq n} 2^{k+h} p_i(k+h) \right) \\
&\leq 2n^2 2^{2n} \sum_{i \leq \log n} ((2n)^i + i) \\
&\leq 2n^2 2^{2n} (\log n)^2 (2n)^{\log n} \\
&\leq 2^{7n}.
\end{aligned}$$

From this fact and the definition of H it follows that, for any x with $|x| = n$, if $x \notin I_n$ then $x \in L^H$ (indeed, if $|x| = n$ and $x \notin I_n$ then, from the definitions of H and L^H, it

follows that $x \in L^H$ iff there is a string y such that $|y| = 8n$ and $xy \notin Q_n(H_{n-1})$. Since $||Q_n(H_{n-1})|| \leq 2^{7n}$ and $||\{xy| \ |y| = 8n\}|| = 2^{8n}$, there exists at least one string y such that $|y| = 8n$ and $xy \notin Q_n(H_{n-1})$, and this implies that $x \in L^H$). Thus, since $||I_n|| \leq 2 \log n$, L^H is a co-sparse set. □

CLAIM 4.

(a) *For each* $n \geq 1$, $i \leq \log n$, *and* x *with* $|x| \leq n$, *it holds that* $M_i^{H_{n-1}}(x) = M_i^H(x)$.

(b) *For each* $n \geq 1$, $i \leq \log n$, *and* x, y *with* $|x|, |y| \leq n$, *it holds that* $xR_i^{H_{n-1}}y \iff xR_i^H y$.

Proof of Claim 4. (a) Note that it suffices to show that $M_i^{H_{n-1}}(x) = M_i^{H_n}(x)$. This is ensured by the fact that, at stage n, no string that is queried during computation $M_i^{H_{n-1}}(x)$ is added to H_{n-1}.

(b) The proof is similar to the proof of (a). □

CLAIM 5. L^H *is* P^H*-immune.*

Proof of Claim 5. From Claim 3, L^H is an infinite set. Let i be an index such that $||L(M_i^H)|| = \infty$. Then a string x exists such that $\log|x| \geq i$ and $M_i^H(x)$ accepts. From part (a) of Claim 4, it also follows that $M_i^{H_{|x|-1}}(x)$ accepts. Thus, a string y exists such that $|y| = |x|$ and $y \in I_{|x|}$. (Since $M_i^{H_{|x|-1}}(x)$ accepts and $x \notin L^{H_{|x|-1}}$, string x is an $M_i^{H_{|x|-1}}$-killer. Thus, a string y must exist such that $|y| = |x|$ and y is a first $M_i^{H_{|x|-1}}$-killer. This implies that $y \in I_{|x|}$.) It follows that $y \notin L^H$ and $M_i^H(y)$ accepts. Hence $L(M_i^H) \not\subseteq L^H$. □

CLAIM 6. *Let* i *be any index. Suppose that, for any* x, y, *it holds that* $x \in L^H$ *and* $xR_i^H y$ *imply that* $y \in L^H$; *then for any* z, w *with* $\log|z| \geq i$, $|w| < |z|$, *and* $w \in L^H$, $wR_i^H z$ *does not hold.*

Proof of Claim 6. Assume that, for any x, y, it holds that $x \in L^H$ and $xR_i^H y$ imply that $y \in L^H$. Let z, w be two strings such that $\log|z| \geq i$, $|w| < |z|$, and $w \in L^H$. Suppose that $wR_i^H z$ holds. Then, by part (b) of Claim 4, it holds that $wR_i^{H_{|z|-1}} z$. Thus, at stage n with $n = |z|$, there exists a string x such that $|x| = n$ and $x \in I_n$ (since z is an $R_i^{H_{n-1}}$-killer.) It follows that, at stage n, no string of the type xy with $|y| = 8n$ is added to H_{n-1}, and thus x is an R_i^H-killer, but this contradicts the assumption. □

CLAIM 7. $L^H \notin \mathrm{KC}^H$.

Proof of Claim 7. The proof follows from Claims 5 and 6. □

Thus, the result is established.

3. Committability. Khadilkar and Biswas [26] introduced *committability*, a notion formalizing the class of sets such that the plausible correctness of a partial proof of membership in the set can be checked via an easily computed query to the set.

DEFINITION 3.1 ([26]).

1. *We call a polynomial-time relation* R *an* eligible defining relation for A *if for no* u *and* v *does* $u\,R\,1v$ *hold, and, for some polynomial* $p(\cdot)$, *it holds that* $x \in A \iff (\exists y)\,[|y| \leq p(|x|) \wedge (y = \epsilon \vee y \in 0\Sigma^*) \wedge xRy]$. *Note that every* NP *set has an eligible defining relation. A string* y *as above is called a* witness for $x \in A$ with respect to the relation R (*or, when the context is clear, simply a* witness.)
2. *A set* A *in* NP *is said to be* committable *if, for some polynomial-time relation* R *that is an eligible defining relation for* A, *there exists a polynomial-time computable function* Witness-check: $\Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$ *such that* $(\forall x, y)$ [Witness-check$(x, y) \in A \iff x \in A$, *and either* y *is a witness for* x *or* $y0$ *is a prefix of a witness for* x *with respect to the eligible relation* R].

Khadilkar and Biswas [26] give examples of committable sets (for example, clearly all NP-complete sets are committable, regardless of paddability) and argue for the naturalness

of their definition. They go on to prove the committability analog of Fortune's work on sparseness [9].

THEOREM 3.2 ([26]). *If a committable set A is $\leq_m^p$-reducible to a co-sparse set, then A is in* P.

We note in passing that one can strengthen this by paralleling Ukkonen's work on sparseness [37] and, with some slight modifications to Yesha's proof, Yesha's work on sparseness [40], obtaining, respectively, the following propositions.

PROPOSITION 3.3. *If A is committable and $A \leq_{\text{disjunctive}}^p$* coSPARSE *then $A \in$* P.

PROPOSITION 3.4. *If A is committable and $A \leq_{\text{bounded-positive-truth-table}}^p$* coSPARSE *then $A \in$* P.

This raises the question of exactly which of the series of results superseding those of Fortune, Mahaney, and Yesha—results such as those of Mahaney, Ogihara, and Watanabe ([31], [34]; see the survey [17])—can also be applied to committability.

The following result puts sharp limits on any such attempted extensions. Theorem 3.6 can be interpreted as saying that Theorem 3.2 cannot be strengthened from "co-sparse" to "sparse" (that is, cannot be strengthened from the analog of Fortune [9] to the analog of Mahaney [31]), unless every infinite FewP $\cap$ coFewP set has an infinite P subset or its complement does (this follows from the fact that all disjunctively self-reducible sets are committable; the limitation thus applies not just to extending the Khadilkar–Biswas claim, but even to any similar claim for disjunctively self-reducible sets). (Definition 3.5 is a standard definition from the literature.)

DEFINITION 3.5.

1. *A partial order $\prec$ on Σ^* is* OK *if there are polynomials p and q such that*
   (a) *every finite $\prec$-decreasing chain is shorter than p of the length of its maximum element, and*
   (b) *$x \prec y$ implies $|x| \leq q(|y|)$ for all $x, y \in \Sigma^*$.*
2. *A set A is said to be* disjunctively self-reducible *if there is a deterministic polynomial-time oracle Turing machine M and an OK partial order such that $L(M^A) = A$ and, on any input x,*
   (a) *M does not query the oracle at all, or if it does, then x is accepted iff any of the queried strings is in the oracle set, and*
   (b) *M asks the oracle only about strings strictly less than x in the OK partial order.*

THEOREM 3.6. *If there is a* P*-bi-immune set in* FewP $\cap$ coFewP *then there is a sparse set in* NP $-$ P (*indeed, even in* FewP $-$ P) *that is disjunctively self-reducible* (*and thus committable*).

Proof. Let L be a P-bi-immune set in FewP $\cap$ coFewP. Since $L \in$ FewP $\cap$ coFewP, it is not hard to see that there exist a polynomial-time relation S, a polynomial-time function h, and a nondecreasing polynomial p such that, for every x,

1. $x \in L \iff (\forall y)\,[y \in W(x) \Rightarrow h(x, y) = 1]$,
2. $x \notin L \iff (\forall y)\,[y \in W(x) \Rightarrow h(x, y) = 0]$,
3. $0 < ||W(x)|| \leq p(|x|)$,

where $W(x) = \{y \mid |y| = p(|x|) \wedge xSy\}$.

Our goal is to define a sparse set T such that, given $x \in T$, it is possible to compute a "witness" for $x \in L$ or $x \in \overline{L}$ (i.e., a string in $W(x)$) making at most a polynomial number of queries to T. One way to do this is to define the set T so that, for every x and y with $|y| \leq p(|x|)$,

$$[x, y] \in T \iff (x \in T \text{ and } y \text{ is a prefix of a string in } W(x)),$$

where $[\cdot, \cdot]$ is the encoding of pairs of strings defined below. The encoding $[\cdot, \cdot]$ will be defined in view of the sparseness of T. Let r be a strictly increasing polynomial such that $(\forall n)\,[r(n) \geq (p(n))^4]$ (without loss of generality, we assume that $(\forall n)\,[p(n) \geq n]$.) For each

pair of strings x and y with $|y| \leq p(|x|)$, define $[x, y] = xy10^{r(|x|)-|xy1|}$. We define the set T as follows: set $\epsilon \notin T$, $0 \in T$, $1 \notin T$, and, for every z with $|z| \geq 2$,

$$z \in T \iff (\exists k \geq 1)(\exists y_1, \ldots, y_k, u_1, \ldots, u_k)\,[z = [\cdots[[0, y_1], y_2]\cdots y_k] \wedge y_1u_1 \in W(0) \wedge y_2u_2 \in W([0, y_1]) \wedge \cdots \wedge y_ku_k \in W([\ldots[[0, y_1], y_2]\ldots y_{k-1}])].$$

From this definition it is clear that, for every x and y with $|y| \leq p(|x|)$,

$$[x, y] \in T \iff (x \in T \wedge yu \in W(x) \text{ for some } u).$$

Thus, if y is not a witness for x (observe that this can be verified in polynomial time) then $[x, y] \in T$ if and only if either $[x, y0] \in T$ or $[x, y1] \in T$. Furthermore, there exists an OK partial order $\prec$ on Σ^* such that, if y is not a witness for x then $[x, y0], [x, y1] \prec [x, y]$. Such a partial order can be defined as follows: for every z and z',

$$z \prec z' \iff (\exists x, y, u)\,[|yu| \leq p(|x|) \wedge z = [x, yu] \wedge z' = [x, y]].$$

Hence, T is disjunctively self-reducible.

Now, we show that $T \in \text{FewP} - \text{P}$. Given $z = [\ldots[[0, y_1], y_2]\ldots y_k]$, the number of accepting paths for z (that is, the number of sequences $u_1, \ldots, u_k$ such that $y_1u_1 \in W(0)$, $y_2u_2 \in W([0, y_1]), \ldots, y_ku_k \in W([\cdots[[0, y_1], y_2]\ldots y_{k-1}])$) is bounded by $p(r^{(0)}(1))p(r^{(1)}(1))\ldots p(r^{(k-1)}(1))$, since, for any i, $||W([\ldots[[0, y_1], y_2]\ldots y_i])|| \leq p(r^{(i)}(1))$, where $r^{(0)}(1) = 1$ and $r^{(i)}(1) = r(r^{(i-1)}(1))$ for $i \geq 1$. It is easy to see, by induction on k, that $p(r^{(0)}(1))p(r^{(1)}(1))\ldots p(r^{(k-1)}(1)) \leq r^{(k)}(1)$. Thus, since $r^{(k)}(1) = |z|$, the accepting paths for z are at most $|z|$, hence $T \in \text{FewP}$.

Suppose that $T \in \text{P}$; then $T \cap L$ and $T \cap \overline{L}$ belong to P. In fact, given $x \in T$, it is possible by prefix search to compute a string $y \in W(x)$; thus it is possible to decide, in polynomial time, whether $x \in L$. Furthermore, since T is an infinite set, either $T \cap L$ or $T \cap \overline{L}$ is an infinite set, contradicting the assumption that L is P-bi-immune.

It remains to show that T is a sparse set. From the definitions of the encoding $[\cdot, \cdot]$ and the set T, it is clear that, for every $n \geq 1$, if $n \notin \{r^{(k)}(1) | k \geq 1\}$ then $T \cap \Sigma^n = \emptyset$. Thus, it suffices to consider $T \cap \Sigma^{r^{(k)}(1)}$ for all $k \geq 0$. Since if $[x, y] \in T \cap \Sigma^{r^{(k)}(1)}$, then $x \in T \cap \Sigma^{r^{(k-1)}(1)}$ and y is a prefix of a string in $W(x)$, it holds that

$$||T \cap \Sigma^{r^{(k)}(1)}|| \leq ||T \cap \Sigma^{r^{(k-1)}(1)}|| \cdot (p(r^{(k-1)}(1)))^2.$$

From this and the fact that $||T \cap \Sigma^{r^{(0)}(1)}|| = 1$, it is easy to show, by induction on k, that $||T \cap \Sigma^{r^{(k)}(1)}|| \leq r^{(k)}(1)$. It follows that $(\forall n)\,[||T \cap \Sigma^n|| \leq n]$. □

Theorem 3.6, which shows that the existence of certain bi-immune sets would imply the existence in NP − P of disjunctively self-reducible sets, should be contrasted with work by Kämper [24] showing that the existence of certain immune sets would imply the existence in NP − P of sets that are *not* disjunctively self-reducible.

Stating the theorem in its more natural contrapositive form, we have the following corollaries.

COROLLARY 3.7. *If all disjunctively self-reducible sparse sets are in* P, *then* FewP ∩ coFewP *is not* P-*bi-immune.*

COROLLARY 3.8. (*Every committable set that* $\leq_m^p$-*reduces to a sparse set is in* P) ⇒ (FewP ∩ coFewP *is not* P-*bi-immune*).

Actually, it is possible to give somewhat stronger evidence for this statement. In fact, we can make the following observation.

Observation 3.9. If there exists a sparse set S in FewP−P then there exists a sparse set S' in FewP−P that is also disjunctively self-reducible.

It suffices to consider the set $S' = \{\langle x, y\rangle | (\exists z : |yz| = p(|x|))\ [xRyz]\}$, where R is a polynomial-time relation (with the associated witness-size polynomial $p(\cdot)$) witnessing that $S \in$ FewP.

The fact that the hypothesis about the existence of P-bi-immune sets in FewP $\cap$ coFewP is stronger than the one about the existence of sparse sets in FewP−P, is ensured by Theorem 3.6 (as the set T constructed in the proof of that theorem belongs to FewP−P). Thus, the value of Theorem 3.6 is to state a somewhat surprising and bizarre connection between P-bi-immune sets in FewP $\cap$ coFewP and sparse sets in FewP−P.

Finally, we mention the following connection between disjunctive self-reducibility, committability, and kernel constructibility.

THEOREM 3.10. *If there is a set that is committable but is not disjunctively self-reducible, then there is a set that is committable and kernel constructible but not disjunctively self-reducible.*

Proof. Let L be a set that is committable but not disjunctively self-reducible. Consider the set $D = L \oplus \Sigma^*$. Clearly, D is still committable but cannot be disjunctively self-reducible (since otherwise L would be disjunctively self-reducible). Furthermore, by Theorem 2.10 it follows that D is also kernel constructible. □

Acknowledgments. We thank Yenjo Han and Benjamin Alexander for many interesting discussions on the uses of Kolmogorov complexity in complexity theory, from which the Kolmogorov complexity use in this paper grew. We thank Jacobo Torán for allowing us to include his Proposition 1.2 and for suggesting Corollary 2.4. We thank Yenjo Han, Johannes Köbler, Mitsunori Ogihara, Uwe Schöning, and Osamu Watanabe for affording us weekend and remote access to printers and libraries that would have been inaccessible without their help. We are very grateful to Vikraman Arvind, Lance Fortnow, Johannes Köbler, Jacobo Torán, Leen Torenvliet, and the anonymous referees for helpful comments, suggestions, and pointers to the literature.

REFERENCES

[1] E. ALLENDER, *Oracles versus proof techniques that do not relativize*, in Proc. 1990 SIGAL International Symposium on Algorithms, Lecture Notes in Computer Science 450, Springer-Verlag, Berlin, New York, Heidelberg, 1990, pp. 39–52.

[2] V. ARVIND AND S. BISWAS, *Kernel constructible languages*, in Record of the 3rd Conference on Foundations of Software Technology and Theoretical Computer Science, National Centre for Software Development and Computing Technique, Tata Institute of Fundamental Research, India, Dec. 1983, pp. 520–538.

[3] ———, *On some bandwidth restricted versions of the satisfiability problem of propositional CNF formulas*, Theoret. Comput. Sci., 68 (1989), pp. 1–14.

[4] L. BERMAN AND J. HARTMANIS, *On isomorphisms and density of NP and other complete sets*, SIAM J. Comput., 6 (1977), pp. 305–322.

[5] H. BUHRMAN, P. VAN HELDEN, AND L. TORENVLIET, *P-selective self-reducible sets: A new characterization of P*, in Proc. 8th Structure in Complexity Theory Conference, IEEE Computer Society Press, 1993, pp. 44–51.

[6] J. CAI, T. GUNDERMANN, J. HARTMANIS, L. HEMACHANDRA, V. SEWELSON, K. WAGNER, AND G. WECHSUNG, *The boolean hierarchy II: Applications*, SIAM J. Comput., 18 (1989), pp. 95–111.

[7] J. CAI, L. HEMACHANDRA, AND J. VYSKOČ, *Promises and fault-tolerant database access*, in Complexity Theory, K. Ambos-Spies, S. Homer, and U. Schöning, eds., Cambridge University Press, London, 1993, pp. 101–146.

[8] S. FENNER, L. FORTNOW, AND S. KURTZ, *An oracle relative to which the isomorphism conjecture holds*, in Proc. 33rd IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1992, pp. 30–39.

[9] S. FORTUNE, *A note on sparse complete sets*, SIAM J. Comput., 8 (1979), pp. 431–433.

[10] R. GAVALDÀ, *Kolmogorov randomness and its applications to structural complexity theory*, Ph.D. thesis, Dept. LSI Universitat Politècnica de Catalunya, Barcelona, Spain, 1992.

[11] R. GAVALDÀ, L. TORENVLIET, O. WATANABE, AND J. BALCÁZAR, *Generalized Kolmogorov complexity in relativized separations*, in Proc. 15th Symposium on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science 452, Springer-Verlag, New York, Berlin, Heidelberg, 1990, pp. 266–276.

[12] J. GROLLMANN AND A. SELMAN, *Complexity measures for public-key cryptosystems*, SIAM J. Comput., 17 (1988), pp. 309–335.

[13] F. HARARY, *A survey of the reconstruction conjecture*, in Graphs and Combinatorics, Lecture Notes in Mathematics 406, Springer-Verlag, New York, Berlin, Heidelberg, 1974, pp. 18–28.

[14] J. HARTMANIS, *Generalized Kolmogorov complexity and the structure of feasible computations*, in Proc. 24th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1983, pp. 439–445.

[15] J. HARTMANIS, R. CHANG, D. RANJAN, AND P. ROHATGI, *Structural complexity theory: Recent surprises*, in Proc. 2nd Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science 447, Springer-Verlag, New York, Berlin, Heidelberg, 1990, pp. 1–12.

[16] L. HEMACHANDRA, A. HOENE, D. SIEFKES, AND P. YOUNG, *On sets polynomially enumerable by iteration*, Theoret. Comput. Sci., 80 (1991), pp. 203–226.

[17] L. HEMACHANDRA, M. OGIHARA, AND O. WATANABE, *How hard are sparse sets?*, in Proc. 7th Structure in Complexity Theory Conference, IEEE Computer Society Press, 1992, pp. 222–238.

[18] L. HEMACHANDRA AND R. SILVESTRI, *Easily checked self-reducibility*, Tech. report TR-431, Department of Computer Science, University of Rochester, Rochester, NY, 1992; revised December 1993.

[19] L. HEMASPAANDRA, S. JAIN, AND N. VERESHCHAGIN, *Banishing robust Turing completeness*, Internat. J. Found. Comput. Sci., 4 (1993), pp. 245–265.

[20] L. HEMASPAANDRA, M. OGIHARA, AND S. TODA, *Space-efficient recognition of sparse self-reducible languages*, Comput. Complexity, 4 (1994), pp. 262–296.

[21] S. HOMER, *On simple and creative sets in NP*, Theoret. Comput. Sci., 47 (1986), pp. 169–180.

[22] J. HOPCROFT, *Recent directions in algorithmic research*, in Proc. 5th GI Conference on Theoretical Computer Science, Lecture Notes in Computer Science 104, Springer-Verlag, New York, Berlin, Heidelberg, 1981, pp. 123–134.

[23] D. JOSEPH AND P. YOUNG, *Self-reducibility: Effects of internal structure on computational complexity*, in Complexity Theory Retrospective, A. Selman, ed., Springer-Verlag, New York, Berlin, Heidelberg, 1990, pp. 82–107.

[24] J. KÄMPER, *A result relating disjunctive self-reducibility to P-immunity*, Inform. Process. Lett., 33 (1990), pp. 239–242.

[25] R. KARP AND R. LIPTON, *Some connections between nonuniform and uniform complexity classes*, in Proc. 12th ACM Symposium on Theory of Computing, 1980, pp. 302–309. An extended version has also appeared as *Turing machines that take advice*, L'Enseignement Mathématique, Ser. 2, 28 (1982), pp. 191–209.

[26] S. KHADILKAR AND S. BISWAS, *Padding, commitment and self-reducibility*, Theoret. Comput. Sci., 81 (1991), pp. 189–199.

[27] D. KRATSCH AND L. HEMACHANDRA, *On the complexity of graph reconstruction*, Math. Systems Theory, 27 (1994), pp. 257–273.

[28] M. LI AND P. VITANYI, *Two decades of applied Kolmogorov complexity*, in Proc. 3rd Structure in Complexity Theory Conference, IEEE Computer Society Press, 1988, pp. 80–101.

[29] ———, *Applications of Kolmogorov complexity in the theory of computation*, in Complexity Theory Retrospective, A. Selman, ed., Springer-Verlag, New York, Berlin, Heidelberg, 1990, pp. 147–203.

[30] T. LONG, *Strong nondeterministic polynomial-time reducibilities*, Theoret. Comput. Sci., 21 (1982), pp. 1–25.

[31] S. MAHANEY, *Sparse complete sets for NP: Solution of a conjecture of Berman and Hartmanis*, J. Comput. System Sci., 25 (1982), pp. 130–143.

[32] S. MAHANEY AND P. YOUNG, *Reductions among polynomial isomorphism types*, Theoret. Comput. Sci., 39 (1985), pp. 207–224.

[33] M. OGIHARA AND A. LOZANO, *On sparse hard sets for counting classes*, Theoret. Comput. Sci., 112 (1993), pp. 255–275.

[34] M. OGIHARA AND O. WATANABE, *On polynomial-time bounded truth-table reducibility of NP sets to sparse sets*, SIAM J. Comput., 20 (1991), pp. 471–483.

[35] H. ROGERS, JR., *The Theory of Recursive Functions and Effective Computability*, McGraw–Hill, New York, Toronto, London, 1967.

[36] A. SHAMIR, *IP=PSPACE*, J. Assoc. Comput. Mach., 39 (1992), pp. 869–877.

[37] E. UKKONEN, *Two results on polynomial time truth-table reductions to sparse sets*, SIAM J. Comput., 12 (1983), pp. 580–587.

[38] L. VALIANT, *The relative complexity of checking and evaluating*, Inform. Process. Lett., 5 (1976), pp. 20–23.

[39] A. YAO, *Coherent functions and program checkers*, in Proc. 22nd ACM Symposium on Theory of Computing, ACM Press, May 1990, pp. 84–94.

[40] Y. YESHA, *On certain polynomial-time truth-table reducibilities of complete sets to sparse sets*, SIAM J. Comput., 12 (1983), pp. 411–425.

SIAM J. COMPUT.
Vol. 24, No. 4, pp. 859–872, August 1995

011

APPROXIMATING THE MINIMUM EQUIVALENT DIGRAPH*

SAMIR KHULLER†, BALAJI RAGHAVACHARI‡, AND NEAL YOUNG§

Abstract. The minimum equivalent graph (MEG) problem is as follows: given a directed graph, find a smallest subset of the edges that maintains all reachability relations between nodes. This problem is NP-hard; this paper gives an approximation algorithm achieving a performance guarantee of about 1.64 in polynomial time. The algorithm achieves a performance guarantee of 1.75 in the time required for transitive closure.

The heart of the MEG problem is the minimum strongly connected spanning subgraph (SCSS) problem—the MEG problem restricted to strongly connected digraphs. For the minimum SCSS problem, the paper gives a practical, nearly linear-time implementation achieving a performance guarantee of 1.75.

The algorithm and its analysis are based on the simple idea of contracting long cycles. The analysis applies directly to 2-EXCHANGE, a general "local improvement" algorithm, showing that its performance guarantee is 1.75.

Key words. directed graph, approximation algorithm, strong connectivity, local improvement

AMS subject classifications. 68R10, 90C27, 90C35, 05C85, 68Q20

1. Introduction.

Connectivity is fundamental to the study of graphs and graph algorithms. Recently, many approximation algorithms for finding minimum subgraphs that meet given connectivity requirements have been developed [1], [9], [11], [15], [16], [24]. These results provide practical approximation algorithms for NP-hard network-design problems via an increased understanding of connectivity properties.

Until now, the techniques developed have been applicable only to *undirected* graphs. We consider a basic network-design problem in *directed* graphs [2], [12], [13], [18], which is as follows: given a digraph, find a smallest subset of the edges (forming a *minimum equivalent graph* (*MEG*)) that maintains all reachability relations of the original graph.

When the MEG problem is restricted to strongly connected graphs we call it *the minimum strongly connected spanning subgraph* (*SCSS*) *problem.* When the MEG problem is restricted to acyclic graphs we call it *the acyclic MEG problem.* The MEG problem reduces in linear time [5] to a single acyclic problem given by the so-called strong component graph, together with one minimum SCSS problem for each strong component (given by the subgraph induced by that component). Furthermore, the reduction preserves approximation in the sense that c-approximate solutions to the subproblems yield a c-approximate solution to the original problem.

Moyles and Thompson [18] observe this decomposition and give exponential-time algorithms for the restricted problems. Hsu [13] gives a polynomial-time algorithm for the acyclic MEG problem.

The related problem of finding a *transitive reduction* of a digraph—a smallest set of edges yielding the same reachability relations—is studied by Aho, Garey, and Ullman [2]. Transitive reduction differs from the MEG problem in that the edges in the transitive reduction are not required to be in the original graph. However, the transitive reduction problem decomposes just like the MEG problem into acyclic and strongly connected instances. For any strongly connected instance, a transitive reduction is given by any Hamilton cycle through the vertices.

*Received by the editors September 7, 1993; accepted for publication (in revised form) April 12, 1994.

†Computer Science Department and Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland 20742 (samir@cs.umd.edu). This research was supported by National Science Foundation Research Initiation Award CCR-9307462.

‡Computer Science Department, University of Texas at Dallas, Richardson, Texas 75083-0688 (rbk@utdallas.edu).

§AT&T Bell Labs, Room 2D-145, 600 Mountain Avenue, Murray Hill, New Jersey 07974. Part of this work was done while at the University of Maryland Institute for Advanced Computer Studies and was supported in part by National Science Foundation grants CCR-8906949 and CCR-9111348 (ney@research.att.com).

For an acyclic instance, the transitive reduction is unique and, as Aho, Garey, and Ullman observe, equivalent to an MEG: it consists of those edges (u, v) for which there is no alternate path from u to v. In fact, Aho, Garey, and Ullman show that the transitive reduction problem is *equivalent* to the transitive closure problem. Thus, the acyclic MEG problem reduces to transitive closure.

The acyclic MEG problem can be solved in polynomial time, whereas the minimum SCSS problem is NP-hard [8], [20]. Consequently, this paper focuses on approximation algorithms for the minimum SCSS problem. By the observations of the preceding paragraphs, the performance guarantees obtained for the minimum SCSS problem carry over to the general MEG problem with the overhead of solving a single instance of transitive closure.

1.1. Our results. Given a strongly connected graph, our basic algorithm finds as long a cycle as it can, contracts the cycle, and recurses. The contracted graph remains strongly connected. When the graph finally collapses into a single vertex, the algorithm returns the set of edges contracted during the course of the algorithm as the desired SCSS.

The algorithm achieves a performance guarantee of any constant *greater* than $\pi^2/6 \approx 1.645$ in polynomial time. We give a nearly linear-time version that achieves a performance guarantee of 1.75. We give examples showing lower bounds on the performance guarantees of the algorithm. For the general algorithm, the lower bounds are slightly above 1.5. For the nearly linear-time version, the lower bound is 1.75, matching the upper bound.

The performance guarantee analysis extends directly to a simple "local improvement" algorithm called 2-EXCHANGE. 2-EXCHANGE starts with the given digraph and performs the following local improvement step as long as it is applicable: find two edges in the current graph that can be replaced by one edge from the original graph, maintaining strong connectivity. Similar local improvement algorithms are natural candidates for many optimization problems but often elude analysis. We prove that the performance guarantee of 2-EXCHANGE is 1.75.

A natural improvement to the cycle-contraction algorithm is to modify the algorithm to solve the problem optimally once the contracted graph has no cycles longer than a given length c. For instance, for $c = 3$ this modification improves the performance guarantee to $\pi^2/6 - 1/36 \approx 1.617$. We use SCSS_c to denote the minimum SCSS problem restricted to digraphs with no cycle longer than c. The minimum SCSS_2 problem is trivial. The minimum SCSS_3 problem can be solved in polynomial time as shown by Khuller, Raghavachari, and Young [14]. However, further improvement in this direction is limited: we show that the minimum SCSS_5 problem is NP-hard. In fact, we show that the minimum SCSS_{17} problem is MAX SNP-hard. This precludes the possibility of a polynomial-time approximation scheme, assuming P$\neq$NP [4].

1.2. Other related work. The union of any incoming branching and any outgoing branching from the same root yields an SCSS with at most $2n - 2$ edges (where n is the number of vertices in the graph). This is a special case of the algorithm given by Frederickson and JáJá [6] that uses minimum weight branchings [7] to achieve a performance guarantee of 2 for weighted graphs. Since any SCSS has at least n edges, this yields a performance guarantee of 2 for the SCSS problem.

Any *minimal* SCSS (one from which no edge can be deleted) has at most $2n - 2$ edges and yields a performance guarantee of 2. The problem of efficiently finding a minimal SCSS is studied by Simon [21]. Gibbons et al. [10] give a parallel algorithm.

A related problem in undirected graphs is to find a smallest subset of the edges forming a biconnected (respectively, bridge-connected (i.e., 2-edge-connected)) spanning subgraph of a given graph. These problems are NP-hard. Khuller and Vishkin [15] give a depth-first-search- (DFS-) based algorithm that achieves a factor of $\frac{5}{3}$ for biconnectivity and $\frac{3}{2}$ for bridge-connectivity. Garg, Santosh, and Singla [9] subsequently improve the approximation

factor for biconnectivity, using a similar approach, to $\frac{3}{2}$. None of these methods appear to extend directly to the minimum SCSS problem.

Undirected graphs having bounded cycle length have bounded tree width. Arnborg, Lagergren, and Seese [3] have shown that many NP-hard problems, including the minimum biconnected-spanning-subgraph problem, have polynomial-time algorithms when restricted to such graphs.

2. Preliminaries. To *contract* a pair of vertices u, v of a digraph is to replace u and v (and each occurrence of u or v in any edge) by a single new vertex and delete any subsequent self-loops and multiedges. Each edge in the resulting graph is identified with the corresponding edge in the original graph or, in the case of multiedges, the single remaining edge is identified with any one of the corresponding edges in the original graph. To contract an edge (u, v) is to contract the pair of vertices u and v. To contract a set S of pairs of vertices in a graph G is to contract the pairs in S in arbitrary order. The contracted graph is denoted by G/S. Contracting an edge is also analogously extended to contracting a set of edges.

Let $\mathcal{OPT}(G)$ be the minimum size of any subset of the edges that strongly connects G. In general, the term cycle refers only to simple cycles.

3. Lower bounds on $\mathcal{OPT}(G)$. We begin by showing that if a graph has no long cycles, then the size of any SCSS is large.

LEMMA 3.1 (Cycle lemma). *For any strongly connected directed graph G with n vertices, if a longest cycle of G has length C, then*

$$\mathcal{OPT}(G) \geq \frac{C}{C-1}(n-1).$$

Proof. Starting with a minimum-size subset that strongly connects the graph, repeatedly contract cycles in the subset until no cycles are left. Observe that the maximum cycle length does not increase under contractions. Consequently, for each cycle contracted, the ratio of the number of edges contracted to the decrease in the number of vertices is at least $\frac{C}{C-1}$. Since the total decrease in the number of vertices is $n-1$, at least $\frac{C}{C-1}(n-1)$ edges are contracted. □

Note that the above lemma gives a lower bound which is existentially tight. For all values of C, there exist graphs for which the bound given by the lemma is equal to $\mathcal{OPT}(G)$. Also note that C has a trivial upper bound of n and, using this, we get a lower bound of n for $\mathcal{OPT}(G)$, which is the known trivial lower bound.

LEMMA 3.2 (Contraction lemma). *For any directed graph G and set of edges S,*

$$\mathcal{OPT}(G) \geq \mathcal{OPT}(G/S).$$

Proof. Any SCSS of G contracted around S (treating the edges of S as pairs) is an SCSS of G/S. □

4. Cycle-contraction algorithm. The algorithm follows. Fix k as any positive integer.

CONTRACT-CYCLES$_k(G)$
1 **for** $i = k, k-1, k-2, \ldots, 2$
2 **while** the graph contains a cycle with at least i edges
3 Contract the edges on such a cycle.
4 **return** the contracted edges

In §6, we will show that the algorithm can be implemented to run in $O(m\alpha(m, n))$ time for the case $k = 3$ and in polynomial time for any fixed value of k, where m is the number

of edges. It is clear that the edge set returned by the algorithm strongly connects the graph. The following theorem establishes an upper bound on the number of edges returned by the algorithm.

THEOREM 4.1. CONTRACT-CYCLES$_k(G)$ *returns at most* $c_k \cdot \mathcal{OPT}(G)$ *edges, where*

$$\frac{\pi^2}{6} \le c_k \le \frac{\pi^2}{6} + \frac{1}{(k-1)k}.$$

Proof. Initially, let the graph have n vertices. Let n_i vertices remain in the contracted graph after contracting cycles with i or more edges ($i = k, k-1, \ldots, 2$).

How many edges are returned? In contracting cycles with at least k edges, at most $\frac{k}{k-1}(n - n_k)$ edges are contributed to the solution. For $i < k$, in contracting cycles with i edges, $\frac{i}{i-1}(n_{i+1} - n_i)$ edges are contributed. Thus the number of edges returned is at most

$$\frac{k}{k-1}(n - n_k) + \sum_{i=2}^{k-1} \frac{i}{i-1}(n_{i+1} - n_i) \le \left(1 + \frac{1}{k-1}\right)n + \sum_{i=3}^{k} \frac{n_i - 1}{(i-1)(i-2)}.$$

Clearly $\mathcal{OPT}(G) \ge n$. For $2 \le i \le k$, when n_i vertices remain, no cycle has more than $i-1$ edges. By Lemmas 3.1 and 3.2, $\mathcal{OPT}(G) \ge \frac{i-1}{i-2}(n_i - 1)$. Thus the number of edges returned, divided by $\mathcal{OPT}(G)$, is at most

$$\frac{\left(1 + \frac{1}{k-1}\right)n}{\mathcal{OPT}(G)} + \sum_{i=3}^{k} \frac{\frac{n_i - 1}{(i-1)(i-2)}}{\mathcal{OPT}(G)} \le \frac{(1 + \frac{1}{k-1})n}{n} + \sum_{i=3}^{k} \frac{\frac{n_i - 1}{(i-1)(i-2)}}{\frac{i-1}{i-2}(n_i - 1)} = \frac{1}{k-1} + \sum_{i=1}^{k-1} \frac{1}{i^2} = c_k.$$

Using the identity (from [17, p. 75]) $\sum_{i=1}^{\infty} 1/i^2 = \pi^2/6$, we get

$$\begin{aligned}
\frac{\pi^2}{6} \le c_k &= \frac{\pi^2}{6} + \frac{1}{k-1} - \sum_{i=k}^{\infty} \frac{1}{i^2} \\
&\le \frac{\pi^2}{6} + \frac{1}{k-1} - \sum_{i=k}^{\infty} \frac{1}{i\,(i+1)} \\
&= \frac{\pi^2}{6} + \frac{1}{k-1} - \frac{1}{k} \\
&= \frac{\pi^2}{6} + \frac{1}{(k-1)k}. \qquad \square
\end{aligned}$$

If desired, standard techniques can yield more accurate estimates of c_k, e.g., $c_k = \pi^2/6 + 1/2k^2 + O(1/k^3)$. If the graph initially has no cycle longer than ℓ ($\ell \ge k$), then the analysis can be generalized to show a performance guarantee of $(k^{-1} - \ell^{-1})/(1 - k^{-1}) + \sum_{i=1}^{k-1} 1/i^2$. For instance, in a graph with no cycle longer than 5, the analysis bounds the performance guarantee (when $k = 5$) by 1.424.

Table 1 gives lower and upper bounds on the performance guarantee of the algorithm for small values of k and in the limit as $k \to \infty$. The lower bounds are shown in the next section.

4.1. Lower bounds on the performance ratio. In this section, we present lower bounds on the performance ratio of CONTRACT-CYCLES$_k(G)$. The graph in Fig. 1 has $\frac{n}{2k-2}$ groups of vertices. Each group consists of a $(2k-2)$-cycle "threaded" with a k-cycle.

In the first iteration, CONTRACT-CYCLES$_k(G)$ can contract the k-cycle within each group, leaving the graph with only 2-cycles. The algorithm subsequently must contract all the remaining edges. Thus, all the $(3k-2)\frac{n}{2k-2} - 2$ edges are in the returned SCSS. The graph contains a Hamilton cycle and the optimal solution is thus n. Hence, for arbitrarily large n, $1 + \frac{k}{2k-2} - 2/n$ is a lower bound on the performance guarantee of CONTRACT-CYCLES$_k(G)$. As k approaches ∞, the lower bound tends to 1.5.

TABLE 1
Bounds on the performance guarantee.

k	Upper bound	Lower bound
3	1.750	1.750
4	1.694	1.666
5	1.674	1.625
∞	1.645	1.500

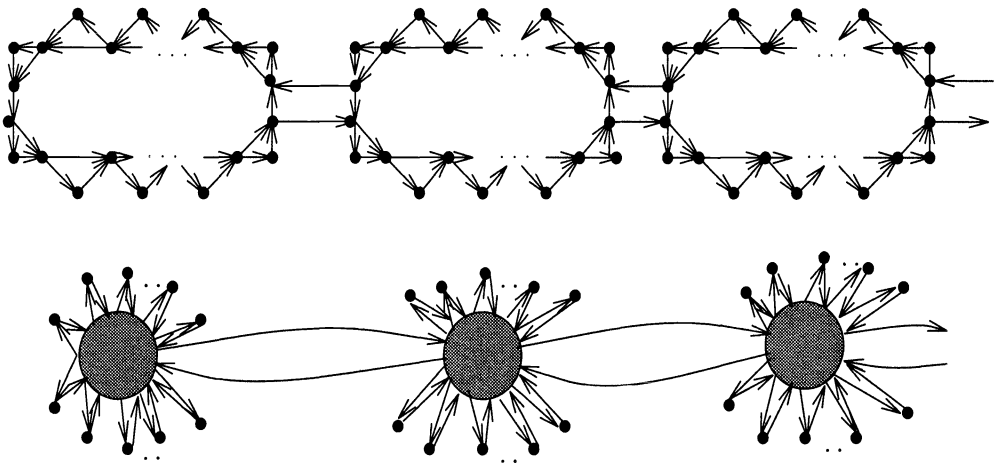


FIG. 1. *Bad example for* CONTRACT-CYCLES$_k(G)$.

5. 2-EXCHANGE algorithm. In this section, we use the cycle-contraction analysis to show that 2-EXCHANGE has a performance guarantee of 1.75. 2-EXCHANGE is a special case of k-EXCHANGE, which is defined as follows.

k-EXCHANGE$(G = (V, E))$ — *Local improvement algorithm.*
1 $E' \leftarrow E$
2 **while** the following improvement step is possible
3 Pick a set E_k of k edges in E' and a set E_{k-1} of up to $k-1$ edges in E such that the set of edges $E'' = (E' - E_k) \cup E_{k-1}$ forms an SCSS.
4 $E' \leftarrow E''$.
5 **return** E'

Note that for fixed k, each step can be performed in polynomial time and reduces the size of E', so k-EXCHANGE runs in polynomial time. The following theorem shows that the approximation factor achieved by 2-EXCHANGE is 1.75.

THEOREM 5.1. *The performance guarantee of* 2-EXCHANGE *is* 1.75.

Proof. We will show that the edges output by 2-EXCHANGE(G) could be output by CONTRACT-CYCLES$_3(G)$. Thus, the performance guarantee of 1.75 for CONTRACT-CYCLES$_3$ carries over to 2-EXCHANGE.

First we show that the performance guarantee is at most 1.75. Let E' be the set of edges returned by 2-EXCHANGE$(G = (V, E))$. Run CONTRACT-CYCLES$_3$ on the graph $G' = (V, E')$. Let H be the set of edges contracted during the first iteration when cycles of at least three edges are contracted. The resulting graph G'/H is strongly connected and has only 2-cycles. Such a graph has a tree-like structure. In particular, an edge (u, v) is present if and only if the reverse edge (v, u) is present.

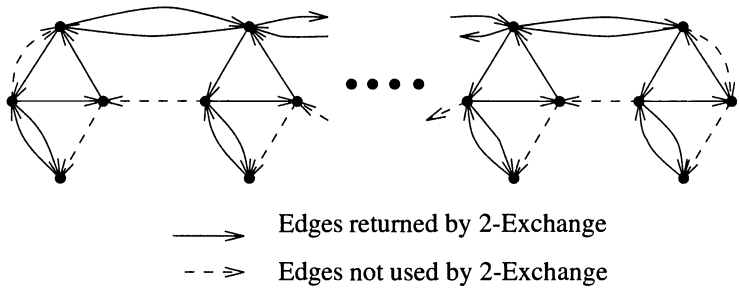


FIG. 2. *Worst-case example for* 2-EXCHANGE.

The important observation is that G/H is equivalent to G'/H. Clearly G'/H is a subgraph of G/H; to prove the converse, suppose that some edge (u, v) of G/H was not in G'/H. Consider adding edge (u, v) to G'/H. By the structure of G'/H, u and v are not adjacent in G'/H, and for each edge on the path from v to u, the reverse edge is also in G'/H. If (u, v) is added to G'/H, these (at least two) reverse edges can be deleted from G'/H without destroying the strong connectivity of G'/H. Consequently, the original edge in G corresponding to (u, v) can be added to G', and the original edges in G' corresponding to the reverse edges can be deleted from G', without destroying the strong connectivity of G'. This contradicts the fact that E' was output by 2-EXCHANGE(G), since E' is eligible for an improvement step.

Next consider executing CONTRACT-CYCLES$_3(G)$. Since G/H is equivalent to G'/H, the sequence of cycles chosen in the first iteration of CONTRACT-CYCLES$_3(G')$ could also be chosen by the first iteration of CONTRACT-CYCLES$_3(G)$. Similarly, the second iteration in CONTRACT-CYCLES$_3(G')$ could be mimicked by CONTRACT-CYCLES$_3(G)$, in which case CONTRACT-CYCLES$_3(G)$ would return the same edge set as CONTRACT-CYCLES$_3(G')$. Since E' is minimal (otherwise an improvement step applies), the edge set returned is exactly E'. Thus, the upper bound on the performance guarantee of CONTRACT-CYCLES$_3$ from Theorem 4.1 is inherited by 2-EXCHANGE.

For the lower bound on the performance guarantee, given the graph in Fig. 2, 2-EXCHANGE can choose a number of edges arbitrarily close to 1.75 times the minimum. There are $\frac{n}{4}$ groups with four vertices in each group. First observe that the graph has a directed Hamilton cycle. The edges marked in Fig. 2 form a solution with which 2-EXCHANGE could terminate. This solution clearly has $\frac{7n}{4}$ edges. This gives the lower bound of 1.75 on the performance of the algorithm. □

6. Implementation. For any fixed k, CONTRACT-CYCLES$_k$ can be implemented in polynomial time using exhaustive search to find long cycles. For instance, if a cycle of size at least k exists, one can be found in polynomial time as follows: For each simple path P of $k-1$ edges, check whether a path from the head of P to the tail exists after P's internal vertices are removed from the graph. If k is even, there are at most $m^{k/2}$ such paths; if k is odd, the number is at most $n\, m^{(k-1)/2}$. It takes $O(m)$ time to decide if there is a path from the head of P to the tail of P. For the first iteration of the for loop, we may have $O(n)$ iterations of the while loop. Since the first iteration is the most time consuming, the algorithm can be implemented in $O(n\, m^{1+k/2})$ time for even k and $O(n^2\, m^{(k+1)/2})$ time for odd k.

6.1. A practical implementation yielding 1.75. Next we give a practical, near linear-time implementation of CONTRACT-CYCLES$_3$. The performance guarantee achieved is $c_3 = 1.75$. CONTRACT-CYCLES$_3$ consists of two phases: (1) repeatedly finding and contracting

cycles of three or more edges (called *long* cycles) until no such cycles exist, and (2) contracting the remaining 2-cycles.

High-level description of the algorithm. To perform phase (1), the algorithm does a DFS of the graph from an arbitrary root. During the search, the algorithm identifies edges for contraction by adding them to a set S. At any point in the search, G' denotes the subgraph of edges and vertices traversed so far. The rule for adding edges to S is as follows: when a new edge is traversed, if the new edge creates a long cycle in G'/S, the algorithm adds the edges of the cycle to S. The algorithm thus maintains that G'/S has no long cycles. When the DFS finishes, G'/S has only 2-cycles. The edges on these 2-cycles, together with S, are the desired SCSS.

Because G'/S has no long cycles and the original graph is strongly connected, G'/S maintains a simple structure.

LEMMA 6.1. *After the addition of any edge to G' and the possible contraction of a cycle by adding it to S,* (i) *the graph G'/S consists of an outward branching and some of its reverse edges,* (ii) *the only reverse edges that might not be present are those on the "active" path: from the supervertex containing the root to the supervertex in G'/S containing the current vertex of the DFS.*

Proof. Clearly the invariant is initially true. We show that each given step of the algorithm maintains the invariant. In each case, if u and w denote vertices in the graph, then let U and W denote the vertices in G'/S containing u and w, respectively.

When the DFS traverses an edge (u, w) to visit a new vertex w, we have the following: Vertex w and edge (u, w) are added to G'. Vertex w becomes the current vertex. In G'/S, the outward branching is extended to the new vertex W by the addition of edge (U, W). No other edge is added and no cycle is created. Thus, part (i) of the invariant is maintained. The supervertex containing the current vertex is now W, and the new "active path" contains the old "active path." Thus, part (ii) of the invariant is also maintained.

When the DFS traverses an edge (u, w) and w is already visited, we have the following: If $U = W$ or the edge (U, W) already exists in G'/S, then no cycle is created, G'/S is unchanged, and the invariant is clearly maintained. Otherwise, the edge (u, w) is added to G' and a cycle with the simple structure illustrated in Fig. 3 is created in G'/S. The cycle consists of the edge (U, W) followed by the (possibly empty) path of reverse edges from W to the lowest common ancestor (lca) of U and W, followed by the (possibly empty) path of branching edges from lca(U, W) to U. Addition of (U, W) to G'/S and contraction of this cycle (in case it is a long cycle) maintains part (i) of the invariant. If the "active path" is changed, it is only because part of it is contracted, so part (ii) of the invariant is maintained.

When the DFS finishes visiting a vertex w, we have the following: No edge is added and no cycle is contracted, so part (i) is clearly maintained. Let u be the new current vertex, i.e., w's parent in the DFS tree. If $U = W$, then part (ii) is clearly maintained. Otherwise, consider the set D of descendants of w in the DFS tree. Since the original graph is strongly connected, some edge (x, y) in the original graph goes from the set D to its complement $V - D$. All vertices in D have been visited, so (x, y) is in G'. By part (i) of the invariant, the vertex in G'/S containing x must be W, while the vertex in G'/S containing y must be U. Otherwise the edge corresponding to (x, y) in G'/S would create a long cycle. □

The algorithm maintains the contracted graph G'/S, using a union-find data structure [22] to represent the vertices in the standard way and three data structures to maintain the branching, the reverse edges discovered so far, and the "active path." When a cycle arises in G'/S, it must be of the form described in the proof of Lemma 6.1 and illustrated in Fig. 3. Using these data structures, the algorithm discovers it and, if it is long, contracts it in a number of union-find operations proportional to the length of the cycle. This yields an $O(m\alpha(m, n))$-time algorithm.

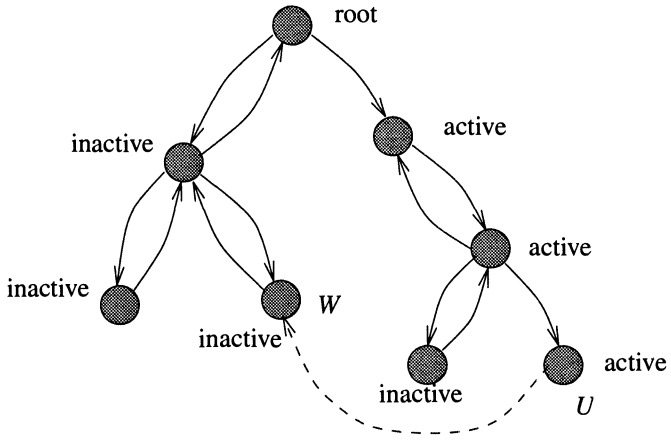


FIG. 3. *Contracted graph G'/S.*

The vertices of G'/S are represented in union-find sets as follows:

MAKE-SET(v). Adds the set $\{v\}$ corresponding to the new vertex of G'/S.

FIND(v). Returns the set in G'/S that contains vertex v.

UNION(u, v). Joins into a single set the two sets corresponding to the vertices in G'/S containing G''s vertices u and v.

The data structures representing the branching, reverse edges, and the active paths, respectively are as follows:

from-root[W]. For each branching edge (U, W) in G'/S, from-root[W] $= (u, w)$ for some $(u, w) \in (U \times W) \cap E$.

to-root[U]. For each reverse edge (U, W) in G'/S, to-root[U] $= (u, w)$ for some $(u, w) \in (U \times W) \cap E$.

to-active[U]. For each vertex U on the "active path" in G'/S, to-active[U] $= (u, w)$, where $(u, w) \in (U \times W) \cap E$ and W is the child of U for which the recursive DFS call is currently executing, unless no recursive DFS is executing, in which case to-active[U] $=$ **current**.

For all other vertices, to-active[U] $=$ **nil**.

Pseudo code for the algorithm is given in Figs. 4 and 5.

By the preceding discussion, the algorithm implements CONTRACT-CYCLES$_3$. It is straightforward to show that it runs in $O(m\alpha(m, n))$ time. Hence, we have the following theorem.

THEOREM 6.2. *There is an $O(m\alpha(m, n))$-time approximation algorithm for the minimum SCSS problem achieving a performance guarantee of* 1.75 *on an m-edge, n-vertex graph.*

Here $\alpha(m, n)$ is the inverse-Ackermann function associated with the union-find data structure [22].

Example to illustrate algorithm. In the example in Fig. 6, the algorithm begins the DFS from vertex 1. It visits vertices 2,3,4 and then traverses the reverse edge (4, 2). Since this edge creates a 3-cycle (2, 3), (3, 4), (4, 2) in G'/S, it contracts the cycle. Next it traverses the reverse edge (3, 1), but does not contract it, since it forms only a 2-cycle in the contracted graph. Continuing the DFS, it visits vertices 5 and 6. When it traverses the edge (6, 4), it discovers and contracts the cycle (3, 1), (1, 5), (5, 6), (6, 4). Next it visits vertices 7 and 8, traversing the reverse edges (8, 7) and (7, 6). Traversing the edge (6, 8), it discovers and

```
CONTRACT-CYCLES3(G = (V, E)) — Pseudocode.
1  S ← {}
2  Choose r ∈ V.
3  DFS(r)
4  Add 2-cycles remaining in G'/S to S.
5  return S

DFS(u) —
1  to-active[FIND(u)] ← current
2  for each vertex w adjacent to u   — traverse edge (u, w) —
3      if (w is not yet visited)   — new vertex —
4        MAKE-SET(w)
5        to-active[FIND(u)] ← from-root[FIND(w)] ← (u, w)
6        DFS(w)
7        to-active[FIND(u)] ← current
8      else   — edge creates cycle in G'/S —
9          if (FIND(u) ≠ FIND(w))   — cycle length at least 2 —
10           (x, y) ← from-root[FIND(u)]
11           if (FIND(x) = FIND(w))   — length two cycle through parent, U − W − U —
12             to-root[FIND(u)] ← (u, w)   — record edge to parent —
13           else
14             (x, y) ← from-root[FIND(w)]
15             if (FIND(x) ≠ FIND(u))   — not a forward edge to child; length of cycle ≥ 3 —
16               CONTRACT-CYCLE(w)
17               S ← S ∪ {(u, w)}
18 to-active[FIND(u)] ← nil
```

FIG. 4. *Practical implementation of* CONTRACT-CYCLES3.

```
CONTRACT-CYCLE(w) —
1  while (to-active[FIND(w)] ≠ current) do
2        if (to-active[FIND(w)] = nil) then   — Go up towards lca along reverse edges. —
3          (c, p) ← to-root[FIND(w)]
4          a ← to-active[FIND(p)]
5        else   — Go down from lca along active path. —
6            (p, c) ← to-active[FIND(w)]
7            a ← to-active[FIND(c)]
           — Contract parent p and child c. —
8        f ← from-root[FIND(p)]
9        t ← to-root[FIND(p)]
10       UNION(p, c)
11       to-active[FIND(w)] ← a
12       from-root[FIND(w)] ← f
13       to-root[FIND(w)] ← t
```

FIG. 5. *Subroutine* CONTRACT-CYCLE.

contracts the 3-cycle (8, 7), (7, 6), (6, 8). In this example, no 2-cycles remain, so it returns just the contracted edges.

7. Potential improvement of CONTRACT-CYCLES$_k$. A natural modification to CONTRACT-CYCLES$_k$ would be to stop when the contracted graph has no cycles of length more than some c and somehow solve the remaining problem optimally.

For instance, for $c = 3$, by following the proof of Theorem 4.1, one can show that this would improve the performance guarantee of CONTRACT-CYCLES$_k$ to $c_k - \frac{1}{36}$ (for $k \geq 4$), matching the lower bound in Table 1. (The lower bound given holds for the modified algorithm.)

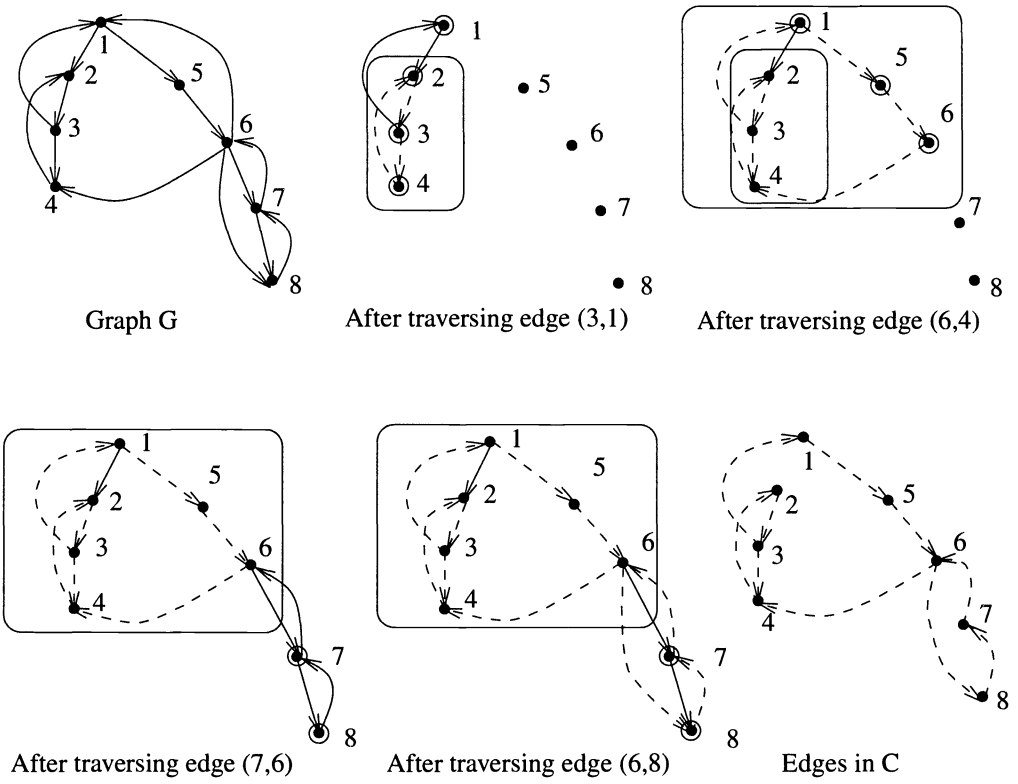


FIG. 6. *Example for illustrating execution of algorithm.*

This leads us to consider the minimum $SCSS_c$ problem—the minimum SCSS problem restricted to graphs with cycle length bounded by c. The following theorem is shown in [14].

THEOREM 7.1. *There is a polynomial-time algorithm for the $SCSS_3$ problem.*

We make no conjecture concerning the $SCSS_4$ problem. However, we next show that the $SCSS_5$ problem is NP-hard and the $SCSS_{17}$ problem is MAX SNP-hard.

7.1. NP-hardness of $SCSS_5$. We prove the following theorem.

THEOREM 7.2. *The minimum $SCSS_5$ problem is NP-hard.*

Proof. The proof is by a reduction from SAT [8]. Fix an arbitrary formula F in conjunctive normal form (CNF). We will build a rooted digraph such that any SCSS contains all the edges out of the root (d of them) and F is satisfiable if and only if there exists an SCSS E' in which each of the remaining $n - 1$ nonroot vertices has out-degree equal to one. Thus the formula will be satisfiable if and only if there is an SCSS with $n - 1 + d$ edges.

The graph has a fixed root vertex r and a vertex for each clause in F (these vertices are not shown in Fig. 7). Each clause vertex has a return edge to the root. For each variable in F, the graph has an instance of the gadget illustrated in Fig. 7. The edges into the gadget come from the root. Each such edge is present in any SCSS. The edges out of the gadget are alternately labeled + and −. For every clause with a positive instance of the variable, one of the + edges goes to the clause vertex. For every clause with a negative instance of the variable, one of the − edges goes to the clause vertex. Unassigned + and − edges go to the root. (The gadget is easily enlarged to allow any number of occurrences.)

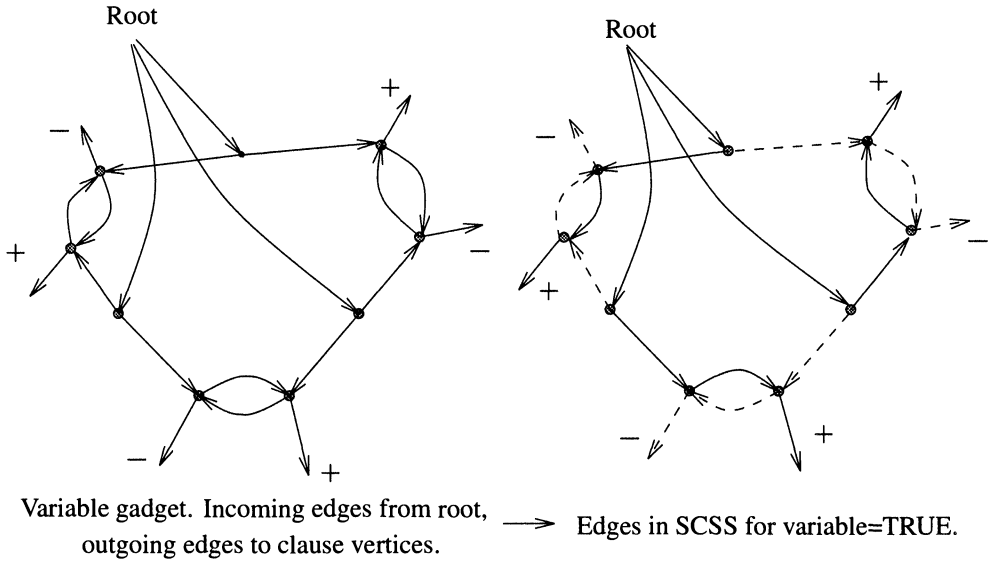


FIG. 7. *Variable gadget for NP-hardness proof.*

The key property of the gadget is that if every (nonroot) vertex has out-degree one in some SCSS, then either all of the counterclockwise edges are in the SCSS (corresponding to the variable being true) or all of the clockwise edges are in the SCSS (corresponding to the variable being false). Thus, given any SCSS of $d + n - 1$ edges, where d is the out-degree of the root and n is the number of vertices in the digraph constructed, it is easy to construct a satisfying assignment for F. Conversely, given any satisfying assignment for F, it is easy to construct an SCSS of size $d + n - 1$. □

7.2. MAX SNP-hardness of SCSS$_{17}$. Next we consider the MAX SNP-hardness of the problem. To prove this we do a reduction from the vertex-cover problem in bounded-degree graphs to the SCSS problem. Since the proof closely follows the reduction from vertex cover to Hamiltonian circuits (see [8]), it is suggested that the reader study this reduction before reading this section. It is known that the problem of finding a minimum vertex cover is MAX SNP-hard in graphs whose maximum degree is bounded by seven [19].

Let G be a connected, undirected graph whose maximum degree is bounded by seven. Let G have m edges and n vertices. We construct a digraph D with $2m + 1$ vertices and no cycle longer than 17. Any vertex cover of G of size s will yield an SCSS in D of size $2m + s$ and vice versa. We then show that, since G has $O(n)$ edges, this yields an L-reduction (i.e., an approximation-preserving reduction [19]).

7.2.1. The construction of D. Applying Vizing's theorem [23], color the edges of G in polynomial time with at most eight colors so that no two edges incident to a vertex share the same color. Let the colors of the edges be chosen from the set $\{1, 2, \ldots, 8\}$.

The construction begins with a special "root" vertex r in D.

As the construction proceeds, each vertex in G will have a "current vertex" in D, initially the root vertex. We process edges in each color class starting with color 1. For each edge (u, v), add a "cover-testing gadget" to D as illustrated in Fig. 8. Specifically, add two new vertices x and y. Add two edges into x: the first, labeled u^+, from the current vertex of u; the

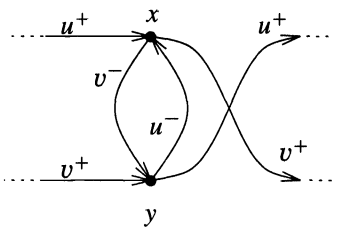


FIG. 8. *A cover-testing component.*

second, labeled u^-, from y. Similarly, add two edges into y: the first, labeled v^+, from the current vertex of v; the second, labeled v^-, from x. Make y the new current vertex of u; make x the new current vertex of v. Finally, after all edges of G have been considered, for each vertex v in G, add an edge labeled v^+ from its final current vertex to the root. The gadgets are implicitly layered with each gadget assigned to a layer corresponding to the color of the associated edge in G.

LEMMA 7.3. *The graph D constructed above has no cycle with more than* 17 *edges.*

Proof. We first assign numbers to the vertices of D. The root r is assigned the number 0. The construction above proceeds in order of increasing color of the edges of G. When considering an edge (u, v) of color c, we add two new vertices: x is added to v's path and y is added to u's path. We assign the vertices x and y the number c. Consider any cycle X of length greater than two in D. It is clear that such a cycle must pass through r, since D is layered. Hence the cycle is of the form $(r, x_1, x_2, \ldots, x_k, r)$. Because we considered the edges in order of increasing color, the numbers assigned to the vertices in X increase at least every two steps in any path in D (not including r). In other words, the numbers assigned to the vertices $x_1, \ldots, x_k$ form a nondecreasing sequence in which no three consecutive vertices get the same number. Since the edges of G were colored with 8 colors, the numbers assigned to the vertices of D range from 0 to 8 (only r gets the number 0). Combining all these, the length of the cycle X is at most 17. □

7.2.2. The analysis. We now show that every vertex cover of G has a corresponding SCSS in D. The proof is similar to the corresponding proof (in the reduction from vertex cover to Hamiltonian circuits) that every vertex cover has a corresponding Hamiltonian circuit. Consider an arbitrary vertex cover S of G. The idea is to choose the paths in the SCSS corresponding to S in D. The paths of the vertices of $V - S$ are not yet connected. Since S forms a vertex cover, the vertices in the paths of $V - S$ can be connected using the cover-testing components.

LEMMA 7.4. *Given a vertex cover of size s in G, an SCSS of D of size $2m + s$ can be constructed.*

Proof. Construct a subgraph H of D as follows. For each vertex u in G, let d_u be the degree of u in G. If u is in the vertex cover, add the $d_u + 1$ edges labeled u^+ in D to H. Otherwise, add the d_u edges labeled u^- in D to H. It is easy to verify that H has the following properties:

1. H has $2m + s$ edges.
2. H has no cycles of length 2.
3. Every vertex of H has at least one outgoing and one incoming edge.

As mentioned earlier, D is layered and every cycle of length greater than 2 contains r. Therefore property 2 above implies that every cycle of H passes through r. By the above conditions, H

contains a path from r to every vertex v and another path from v to r, and is therefore strongly connected. To obtain a path from any v to r, start from v and keep traversing an outgoing edge (which exists by property 3) from the current vertex. Such a path must eventually reach r because r is contained in every cycle of H. Hence H satisfies the lemma. □

We now show that every SCSS of D corresponds to a vertex cover of G. The proof works by showing that any SCSS can be converted into a "canonical" SCSS, whose size is no larger, that corresponds to a vertex cover of G.

LEMMA 7.5. *Given an SCSS in D of size $2m + s$, a vertex cover of G of size s can be constructed.*

Proof. As long as some nonroot vertex y has both of its incoming edges in the SCSS, modify the SCSS as follows: Let (x, y) be the edge labeled v^- for some v. Remove the edge (x, y) and add the other edge out of x if it is not already present. Alternatively, if some nonroot vertex x has both of its outgoing edges in the SCSS, remove the edge (x, y) and add the other edge into y. Repeat either modification as long as applicable.

By the layering of D, each modification maintains the strong connectivity of the SCSS. Clearly none of the modifications increase the size. Each step reduces the number of edges labeled u^- for some u in the SCSS, so after at most $2m$ steps neither modification applies, and in the resulting SCSS every nonroot vertex has exactly one incoming edge and one outgoing edge in the SCSS.

An easy induction on the layering shows that for any vertex v in G, either all of the edges labeled v^+ in D are in the SCSS or none are in the SCSS, in which case all of the edges labeled v^- are in the SCSS. Let C be the set of vertices in G of the former kind. It is easy to show that the size of the SCSS is $2m + |C|$, so that $|C| \leq s$. For every edge (u, v) in G, the form of the gadget ensures that at least one of the two endpoints is in C. Hence, C is the desired cover. □

THEOREM 7.6. *The minimum $SCSS_{17}$ problem is MAX SNP-hard.*

Proof. Let G be an arbitrary undirected graph G whose maximum degree is bounded by seven. Let G have m edges and n vertices. Construct the digraph D as shown earlier. By Lemma 7.3, D has no cycles greater than 17. By Lemma 7.4, any vertex cover in G of size s can be used to obtain an SCSS of D of size $2m + s$. Conversely, by Lemma 7.5 an SCSS of D of size $2m + s$ can be used to obtain a vertex cover of G of size s. Since the degree of G is bounded, $m = O(n) = O(s)$ and it is easily verified that this yields an L-reduction from degree-bounded vertex cover to the minimum $SCSS_{17}$ problem. □

8. Open problems. An obvious problem is to further characterize the various complexities of the minimum $SCSS_k$ problems.

The most interesting open problem is to obtain a performance guarantee that is less than 2 for the weighted strong connectivity problem (as mentioned earlier, the performance factor of 2 is from Frederickson and JáJá [6]). Such an algorithm may have implications for the weighted 2-connectivity problem [15] in undirected graphs as well.

The performance guarantee of k-EXCHANGE probably improves as k increases. Proving this would be interesting—similar "local improvement" algorithms are applicable to a wide variety of problems.

Acknowledgments. We thank the referees for useful comments.

REFERENCES

[1] A. AGRAWAL, P. KLEIN, AND R. RAVI, *When trees collide: An approximation algorithm for the generalized Steiner problem on networks*, Proc. 23rd ACM Symposium on Theory of Computing, New Orleans, LA, 1991, pp. 134–144.

[2] A. V. AHO, M. R. GAREY, AND J. D. ULLMAN, *The transitive reduction of a directed graph*, SIAM J. Comput., 1 (1970), pp. 131–137.

[3] S. ARNBORG, J. LAGERGREN, AND D. SEESE, *Easy problems for tree-decomposable graphs*, J. Algorithms, 12 (1991), pp. 308–340.

[4] S. ARORA, C. LUND, R. MOTWANI, M. SUDAN, AND M. SZEGEDY, *Proof verification and hardness of approximation problems*, Proc. 33rd IEEE Symposium on Foundations of Computer Science, Pittsburgh, PA, 1992, pp. 14–23.

[5] T. H. CORMEN, C. E. LEISERSON, AND R. L. RIVEST, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1989.

[6] G. N. FREDERICKSON AND J. JÁJÁ, *Approximation algorithms for several graph augmentation problems*, SIAM J. Comput., 10 (1981), pp. 270–283.

[7] H. N. GABOW, Z. GALIL, T. SPENCER, AND R. E. TARJAN, *Efficient algorithms for finding minimum spanning trees in undirected and directed graphs*, Combinatorica, 6 (1986), pp. 109–122.

[8] M. R. GAREY AND D. S. JOHNSON, *Computers and intractability: A guide to the theory of NP-completeness*, W. H. Freeman, San Francisco, CA, 1979.

[9] N. GARG, V. SANTOSH, AND A. SINGLA, *Improved approximation algorithms for biconnected subgraphs via better lower bounding techniques*, Proc. 4th Annual ACM–SIAM Symposium on Discrete Algorithms, Austin, TX, 1993, pp. 103–111.

[10] P. GIBBONS, R. M. KARP, V. RAMACHANDRAN, D. SOROKER, AND R. E. TARJAN, *Transitive compaction in parallel via branchings*, J. Algorithms, 12 (1991), pp. 110–125.

[11] M. GOEMANS AND D. WILLIAMSON, *A general approximation technique for constrained forest problems*, Proc. 3rd Annual ACM–SIAM Symposium on Discrete Algorithms, Orlando, FL, 1992, pp. 307–316.

[12] F. HARARY, R. Z. NORMAN, AND D. CARTWRIGHT, *Structural models: An introduction to the theory of directed graphs*, John Wiley, New York, NY, 1965.

[13] H. T. HSU, *An algorithm for finding a minimal equivalent graph of a digraph*, J. Assoc. Comput. Mach., 22 (1975), pp. 11–16.

[14] S. KHULLER, B. RAGHAVACHARI, AND N. YOUNG, *On strongly connected digraphs with bounded cycle length*, UMIACS-TR-94-10/CS-TR-3212, University of Maryland, 1994.

[15] S. KHULLER AND U. VISHKIN, *Biconnectivity approximations and graph carvings*, Proc. 24th ACM Symposium on Theory of Computing, 1992, pp. 759–770; J. Assoc. Comput. Mach., 41 (1994), pp. 214–235.

[16] P. N. KLEIN AND R. RAVI, *When cycles collapse: A general approximation technique for constrained two-connectivity problems*, Proc. 3rd Integer Programming and Combinatorial Optimization Conference, Mathematical Programming Society, Erice, Italy, 1993, pp. 39–56.

[17] D. E. KNUTH, *Fundamental Algorithms*, Addison–Wesley, Menlo Park, CA, 1973.

[18] D. M. MOYLES AND G. L. THOMPSON, *An algorithm for finding the minimum equivalent graph of a digraph*, J. Assoc. Comput. Mach., 16 (1969), pp. 455–460.

[19] C. H. PAPADIMITRIOU AND M. YANNAKAKIS, *Optimization, approximation, and complexity classes*, J. Comput. System Sci., 43 (1991), pp. 425–440.

[20] S. SAHNI, *Computationally related problems*, SIAM J. Comput., 3 (1974), pp. 262–279.

[21] K. SIMON, *Finding a minimal transitive reduction in a strongly connected digraph within linear time*, in Proc. 15th International Workshop WG'89, Lecture Notes Comput. Sci. 411, Springer-Verlag, 1989, pp. 245–259.

[22] R. E. TARJAN, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[23] V. G. VIZING, *On an estimate of the chromatic class of a P-graph*, Diskret. Anal., 3 (1964), pp. 25–30. (In Russian.)

[24] D. P. WILLIAMSON, M. X. GOEMANS, M. MIHAIL, AND V. V. VAZIRANI, *A primal-dual approximation algorithm for generalized Steiner network problems*, Proc. 25th ACM Symposium on Theory of Computing, San Diego, CA, 1993, pp. 708–717.

SIAM J. COMPUT.
Vol. 24, No. 4, pp. 873–921, August 1995

012

FIXED-PARAMETER TRACTABILITY AND COMPLETENESS I: BASIC RESULTS*

ROD G. DOWNEY† AND MICHAEL R. FELLOWS‡

Abstract. For many fixed-parameter problems that are trivially soluable in polynomial time, such as (k-)DOMINATING SET, essentially no better algorithm is presently known than the one which tries all possible solutions. Other problems, such as (k-)FEEDBACK VERTEX SET, exhibit *fixed-parameter tractability*: for each fixed k the problem is soluable in time bounded by a polynomial of degree c, where c is a constant independent of k. We establish the main results of a completeness program which addresses the apparent fixed-parameter intractability of many parameterized problems. In particular, we define a hierarchy of classes of parameterized problems $FPT \subseteq W[1] \subseteq W[2] \subseteq \cdots \subseteq W[SAT] \subseteq W[P]$ and identify natural complete problems for $W[t]$ for $t \geq 2$. (In other papers we have shown many problems complete for $W[1]$.) DOMINATING SET is shown to be complete for $W[2]$, and thus is not fixed-parameter tractable unless INDEPENDENT SET, CLIQUE, IRREDUNDANT SET, and many other natural problems in $W[2]$ are also fixed-parameter tractable. We also give a compendium of currently known hardness results as an appendix.

Key words. fixed-parameter tractable, W-hierarchy, parameterized complexity, DOMINATING SET, t-NORMALIZED SATISFIABILITY

AMS subject classifications. Primary, 03D15, 68Q15, 68Q25; Secondary, 03D10, 03D20, 03D30, 03D80, 68R10

1. Introduction.

Many natural computational problems have input that consists of a pair of items. For example, the GRAPH GENUS problem is that of determining for an input pair (G, k), where G is a graph and k is a positive integer, whether the graph G embeds on the genus k surface. The problem of MINOR TESTING is that of determining for an input pair of graphs (G, H) whether H is a minor of G.

One of the reasons for our interest in parameterized problems is that while many of these problems are NP-complete $PSPACE$-complete, or even provably intractable, it is sometimes the case that only a small range of parameter values are really important in practice, so that the (apparent) intractability of the general problem may be unduly pessimistic information. For many parameterized problems, we now have encouraging and perhaps useful fixed-parameter tractability results, such as the following.

THEOREM 1.1 (Robertson and Seymour [108]). *For every fixed graph H it can be determined in time $O(n^3)$ whether a graph G of order n has a minor isomorphic to H.*

THEOREM 1.2 (Bienstock and Monma [15]). *For every fixed k, it can be determined in time $O(n)$ whether a graph G of order n can be embedded in the plane so that k faces cover all the vertices.*

THEOREM 1.3 (Bodlaender [16]). *For every fixed k, it can be determined in time $O(n)$ whether a graph G of order n has a spanning tree with at least k leaves.*

THEOREM 1.4 (Lagergren [95]). *For every fixed k, it can be determined in time $O(n \log^2 n)$ whether a graph G of order n has treewidth at most k.*

THEOREM 1.5 (Plehn and Voigt [102]). *For every fixed graph H of treewidth w, it can be determined in time $O(n^{w+1})$ whether a graph G of order n has a subgraph isomorphic to H. (Note that here the parameter is (a coding of) H.)*

*Received by the editors March 24, 1992; accepted for publication (in revised form) August 1, 1994. Preliminary versions of some of the results of this paper were presented at the 21st Manitoba Conference on Numerical Mathematics and Computing, Winnipeg, Manitoba, Canada, October 1991. These were reported in [55].

†Mathematics Department, Victoria University, P.O. Box 600, Wellington, New Zealand. The research of this author was supported by a grant from Victoria University IGC and by the United States–New Zealand Cooperative Science Foundation grant INT 90-20558.

‡Department of Computer Science, University of Victoria, Victoria, British Columbia, V8W 3P6, Canada. The research of this author was supported by Office of Naval Research contract N00014-88K-0456, National Science Foundation grant MIP-8919312, and the National Science and Engineering Research Council of Canada.

THEOREM 1.6 (Fellows and Langston [74]). *For every fixed k, it can be determined in time $O(n)$ whether a graph G of order n has a cycle of length at least k.*

THEOREM 1.7 (Bodlaender [17], Downey and Fellows [55]). *For every fixed k, it can be determined in time $O(n)$ whether a graph G of order n contains k vertex-disjoint cycles.*

For other parameterized problems, such as DOMINATING SET (given a graph G and a positive integer k is there a set of k vertices in G such that every vertex either belongs to the set or has a neighbour in the set) we have the contrasting situation where essentially no better algorithm is known than the "trivial" one which just exhaustively tries all possible solutions. For each fixed k, k- DOMINATING SET is soluable in this way in time $O(n^{k+1})$.

We make the following definitions in order to frame these complexity issues.

DEFINITION 1.8. *A* parameterized problem *is a set $L \subseteq \Sigma^* \times \Sigma^*$ where Σ is a fixed alphabet.*

For a parameterized problem L and $y \in \Sigma^*$ we write L_y to denote the associated fixed-parameter problem (y is the parameter) $L_y = \{x | (x, y) \in L\}$. We refer to L_y as the yth *slice* of L.

DEFINITION 1.9. *A parameterized problem L is* ((weakly) uniformly) fixed-parameter tractable *if there exists a constant α and an algorithm to determine if (x, y) is in L in time $f(|y|) \cdot |x|^{\alpha}$, where $f : N \to N$ is an arbitrary function. If f is recursive then we say that L is* strongly uniformly fixed-parameter tractable. *Finally, we say that L is nonuniformly f.p. tractable if there is a family of algorithms $\{\Psi_x : x \in N\}$ and a function f such that Ψ_x determines if (x, y) is in L in time $f(|y|) \cdot |x|^{\alpha}$.*

In recent years a variety of methods useful for demonstrating fixed-parameter tractability have emerged, such as the well-quasiordering results of Robertson and Seymour [106], [107], [108], and general algorithmic methods for bounded treewidth (e.g., Abrahamson and Fellows [4], [5]; Arnborg [7]; Arnborg, Lagergren, and Seese [10]; Bern, Lawler, and Wong [13]; Courcelle [49]; and Wimer, Hedetniemi, and Laskar [118]).

The reader should note an important detail of the definition of fixed-parameter tractability given above. The results of Theorems 1.2–1.7 (and our Theorem 2.1 below) are clearly uniform; the proofs of these results can be implemented as a single algorithm that works for every parameter value. Consider, contrastingly, the consequence of Theorem 1.1 and the graph minor theorem [108] that for each fixed k, it can be determined in time $O(n^3)$ whether a graph G of order n embeds on the surface of genus k. It is not immediately clear how these (infinitely many) distinct $O(n^3)$ algorithms, each based on a different finite obstruction set, can be combined into a single finite algorithm. This can done, however, by the two different methods of Fellows and Langston [74], [75]. Almost all of the known fixed-parameter tractability results are (or can be made) uniform. While it is possible to construct examples (Downey and Fellows [58]) that show that the notions of tractability are indeed provably distinct, we also remark that there are natural examples of apparently all flavours of tractability. For instance, consider the following examples.

PLANAR IMPROVEMENT
Instance: A graph G.
Parameter: k.
Question: Is G a subgraph of a planar graph G' of diameter at most k?

GRAPH LINKING NUMBER
Instance: A graph G.
Parameter: k.
Question: Can G be embedded in 3-space so that at most k disjoint cycles are topologically linked?

We remark that both of these examples are known to be fixed-parameter tractable via the Robertson–Seymour theorems. But at present PLANAR IMPROVEMENT is known only to be weakly uniformly tractable and GRAPH LINKING NUMBER is known only to be nonuniformly tractable.

The difference between the known fixed-parameter complexity of DOMINATING SET and the problems addressed in Theorems 1.1–1.7 and the two examples, is analogous to the apparent complexity difference between *NP*-complete problems and problems in P. For most *NP*-complete problems we essentially know no better algorithm than the "trivial" one requiring exponential time which tries all possible solutions.

If $P = NP$ then DOMINATING SET is fixed-parameter tractable. A converse to this statement is not known and is perhaps unlikely. The reasonable (we think) conjecture that DOMINATING SET is not fixed-parameter tractable is thus apparently stronger than the conjecture that $P \neq NP$. Certainly there is oracle evidence to perhaps support this claim (Downey and Fellows [58]). The graph minor theorem has the consequence that for each fixed surface we can decide graph embedability by employing *finitely many* minor tests. Thus the fixed-parameter tractability of MINOR TESTING leads to the fixed-parameter tractability of the GRAPH GENUS problem. This may be kept in mind as a motivating example for the following definition.

DEFINITION 1.10 (uniform reduction). *A* (uniform) reduction *of a parameterized problem L to a parameterized problem L' is an oracle algorithm A that on input (x, y) determines whether $x \in L_y$ and satisfies*

(1) *There is an arbitrary function $f : N \to N$ and a polynomial q such that the running time of A is bounded by $f(|y|)q(|x|)$.*

(2) *For each $y \in \Sigma^*$ there is a finite subset $J_y \subseteq \Sigma^*$ such that A consults oracles only for fixed-parameter decision problems L'_w where $w \in J_y$.*

Of course in the above an oracle computation takes only one unit of time. If the oracle is consulted only once by A, then we will term the reduction *many:*1. As with the notion of tractability, there is a strong version. If the function f and the map taking y to J_y are both recursive, we say that the reduction is *strongly uniform.* (Similarly there is a notion of nonuniform reduction, which we do not consider in detail here.) All of the results we prove in this paper hold for all of the frameworks, with the single exception of Theorem 4.1.

LEMMA 1.11. *If the parameterized problem L reduces to the parameterized problem L', and if L' is fixed-parameter tractable, then L is fixed-parameter tractable.*

Proof. Let $f(|y|)q(|x|)$ be the bound on the running time of the reduction from L to L', and suppose L'_w is decidable in time $g(|w|) \cdot n^\alpha$. Without loss of generality, we can take f and g to be increasing. Let $y \in \Sigma^*$ and let $J_y \subseteq \Sigma^*$ be the associated finite subset of Σ^* for the reduction. Then we can determine if $(x, y) \in L$ in time $O(f(|y|)q(|x|)g(m)(f(|y|)q(|x|)^\alpha)$ where $m = \max\{|w| : w \in J_y\}$. □

Working definition. Actually for the sake of most naturally occuring concrete reductions, we can take a simpler definition than the above. Most concrete reductions are m-reductions of the form $(x, k) \mapsto (x', f(k))$ with x' depending upon x and k, running in time $h(k)|x|^\alpha$ with f and h recursive. The reason for this is that most natural problems are smooth in the sense that one has a natural encoding of the slices with parameters below k into the kth slice and hence we only need to look at one slice $f(k)$ for any input (x, k). The general definition is useful at times, and is certainly needed for structural theorems. It does the reader no harm to take the simplified definition for the remainder of the paper.

Remark: Alternative definition. Another view of the ideas above is provided by Cai et al. [41], the *advice* view. In that paper, Cai et al. prove that if L is a parameterized language,

then $L \in FPT$ iff there is a function $f : N \to \Sigma^*$ (the advice function) and a P-time oracle (deterministic) Turing machine Φ, such that for all (x, k),

$$(x, k) \in L \text{ iff } \Phi^{f(k)}(x) \text{ accepts.}$$

That is, if we allow for each k a finite piece of advice, then we can solve all the instances in time $|x|^\alpha$. There is similarly a relativized version for the reductions. We refer the reader to Cai et al. [41] for more details.

The plan of this paper is as follows. In §2 we prove a particular combinatorial problem reduction that plays a key role in our main theorem, which is presented in §3. Section 4 summarizes our results and discusses open problems. During the time this paper was under consideration many new hardness and completeness results have been found, and we see that this theory seems to have wide-ranging consequences for the classification of parameterized hardness results. This is particularly true of, for instance, problems in molecular biology where one is often interested in a small parameter (e.g., the number of strands of DNA) yet the problem is very large (e.g., the length of the strand), and in very large scale integration design where we might have a small number of wafers of very large size. In the appendix we give a list of currently known hardness results, as well as a list of open classification problems.

Related work and historical remarks. To conclude this section we give some brief remarks concerning related investigations. As far as we are aware, the first person to suggest that something might be interesting in the fact that DOMINATING SET seems to require time $\Omega(n^{k+1})$ was Ken Regan [104], in some comments in that paper. Regan did not pursue this issue. There have been investigations into "nondeterminism in P" such as the Kintala–Fischer β-hierarchy [93] and the work of Buss and Goldsmith [35] but these and similar related investigations were mainly structural, and looked at problems for a single k. One then cannot use P-time reductions since the class is in $DTIME(n^{f(k)})$ and usually $DTIME(n^k)$. These authors instead used small reductions such as *quasilinear* (e.g., Gurevich and Shelah [86]) time reductions, which are of course machine dependent. The only paper to truely study the *asymptotic* parameterized behavior (i.e., the issue of n^α versus $n^{f(k)}$ with $f(k) \to \infty$) is Abrahamson et al. [3]. (Some of these results were recast in [35].) In that paper the objects were P-checkable, P-indexed relations, called (polynomial time) *generator tester* pairs. For each k, one needed to be able to generate a P-time list of potential candidates for solutions that were easy to test. The actual definition is very involved but the following example gives the flavour.

If we consider the problem VERTEX COVER, then for an instance G and a parameter k, the potential witnesses would be pairs consisting of G and a vertex cover V with the index of $V \leq j = \binom{n}{0} + \cdots + \binom{n}{k}$. The ideas only seemed to apply to relations in NP. While there is a notion of parameterized tractability in [3], it is roughly equivalent to our notion of nonuniform fixed-parameter tractability and hence suffers from the problem that the tractable classes can be nonrecursive. The real problem with that paper is that the notion of reducibility, which is defined on relations rather than parameterized languages, is rather unnatural and very unwieldy, and the notion of intractability is that of (essentially) being P-complete (or "dual P-complete") under *logspace* reducibility "by the slice." That is, in [3], problems are PGT-complete (in their notation) only when for each k they are more or less P-complete. Of course this means that the Abrahamson et al. [3] ideas cannot address things such as DOMINATING SET and INDEPENDENT SET, nor apparently anything in the $W[t]$ classes below. We remark that the Abrahamson et al. [3] results can be easily placed in our setup because they give $W[P]$-completeness results (see, e.g., Abrahamson, Downey, and Fellows [1], [2]). We see our major contributions as identifying the correct notions of reducibility, identifying the correct setting for the study of parameterized intractibility, and finally identifying some "good" problems

with which to measure hardness. The number of problems that have now been identified as $W[1]$ hard, and hence apparently intractable, would seem to support our claims. ($W[1]$ hardness is not analysed in the present paper, but in Downey and Fellows [56].) At this stage we will only remark that quite a number of parameterized problems have been shown to be $W[1]$-complete. (See the appendix.) We also have a sort of Cook–Levin theorem for $W[1]$ which we feel strongly suggests its intractability. Cai et al. [40] have shown that the following very generic problem is $W[1]$ complete.

SHORT TURING MACHINE ACCEPTANCE
Input: A nondeterministic Turing machine M and a string x.
Parameter: k.
Question: Does M have an accepting computation for x in k or fewer steps?

To conclude, we reiterate the remark that the notion of parameterized tractability is *not* a refinement of the classical notions arising from *NP*-completeness, despite the fact that many of our examples arise from this arena. Problems can be *PSPACE* complete (e.g., ALTERNATING HITTING SET; see Abrahamson, Downey, and Fellows [2]) and yet have parameterized versions that are FPT. Also there are problems that are almost certainly not *NP*-complete unless some unlikely collapse occurs such as *NP* to *LOGNP* and yet their parameterized versions can be $W[2]$ or $W[1]$ hard (e.g., the VAPNIK CHERVONENKIS DIMENSION; see Downey, Evans, and Fellows [54], Downey and Fellows [59], and Papadimitriou and Yannakakis [100]). Finally take any set A and consider the parameterized problem $\{(x, x) : x \in A\}$. Then this problem is just as hard as A classically, so it can even be provably intractable, and yet it is trivially FPT. These examples show that the parameterized complexity of problems and their unparameterized versions are pretty well unrelated, and thus our investigations point to a new dimension in the structure of problems.

2. A key combinatorial reduction. Neither of the well-known computational problems of (1) determining whether a graph G has a dominating set of size k (DOMINATING SET), and (2) determining whether a graph G has an independent set of size k (INDEPENDENT SET) is know to be fixed-parameter tractable, and it is perhaps a reasonable conjecture that they are not. The reader skeptical of this conjecture and willing to challenge it, will be advised by the results of this section to begin by working on INDEPENDENT SET, since a consequence of Theorem 2.1 is that INDEPENDENT SET reduces to DOMINATING SET (and so the latter is "apparently harder" with respect to fixed-parameter tractability). Presently the best known results for these problems are the trivial $O(n^{k+1})$ algorithm for DOMINATING SET and a nontrivial algorithm for INDEPENDENT SET due to Nesetril and Poljak [99], requiring time $O(n^{k(2+\epsilon)/3})$, where $2+\epsilon$ represents the best known exponent for fast matrix multiplication.

We show that WEIGHTED CNF SATISFIABILITY (defined below) reduces to DOMINATING SET. By the *weight* of a truth assignment to a set of boolean variables, we mean the number of variables assigned the value *true*, in the same way that the weight of a binary vector means the number of 1's in the vector. Since INDEPENDENT SET (and many other parameterized problems) easily reduce to this problem, we have the consequence claimed above. For example, a graph $G = (V, E)$ has a k-element independent set if and only if the expression $\prod_{uv \in E}(\bar{u} + \bar{v})$ has a weight k truth assignment. The notion of reduction that we use is (the working reduction) defined in §1.

WEIGHTED CNF SATISFIABILITY
Instance: A boolean expression X in conjuctive normal form.
Parameter: k.
Question: Is there a truth assignment of weight k that satisfies X?

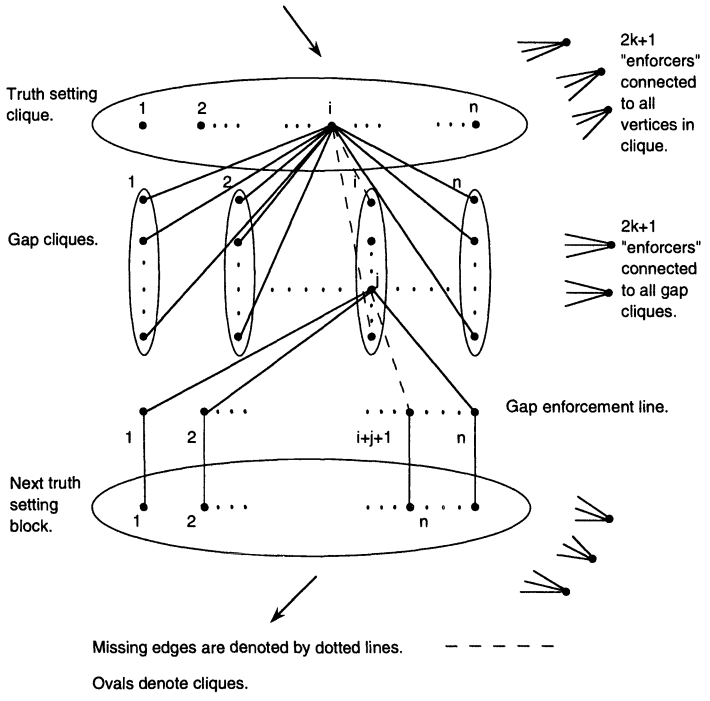


FIG. 1. *Gadget for CNFSAT* $\leq_{f.p.}$ *DOMINATING SET.*

THEOREM 2.1. *WEIGHTED CNF SATISFIABILITY strongly uniformly reduces to DOMINATING SET.*

Proof. Let X be a Boolean expression in conjuctive normal form consisting of m clauses $C_1, \ldots, C_m$ over the set of n variables $x_0, \ldots, x_{n-1}$. We show how to produce in polynomial-time by local replacement a graph $G = (V, E)$ that has a dominating set of size $2k$ if and only if X is satisfied by a truth assignment of weight k.

A diagram of the gadget used in the reduction is given in Fig. 1. The idea of the proof is as follows. There are k of the gadgets arranged in a circle. Each of the gadgets has three main parts. Taken clockwise from top to bottom, these are variable selection, gap selection, and gap enforcement. The variable selection component is a clique and the gap selection consists of n cliques which we call columns. Our first action is to ensure that in *any* dominating set of $2k$ elements, we must pick one vertex from each of these two components. This goal is achieved by the $2k$ sets of $2k + 1$ enforcers, vertices from V_4 and V_5. (The names refer to the sets below.) Take the set V_4, for instance. For a fixed r, these $2k + 1$ vertices are connected to all of the variable selection vertices in the component $A(r)$, and nowhere else. Thus if they are to be dominated by a $2k$ dominating set, then we must choose *some* element in the set $A(r)$, and similarly we must choose an element in the set $B(r)$ by virtue of the V_5 enforcers. Since we will need exactly $2k$ (or even $\leq 2k$) dominating elements it follows that we must pick *exactly* one from each of the $A(r)$ and $B(r)$ for $r = 1, \ldots, k$.

As the name suggests these will be picked by the variable selection components, $A(r)$, $r = 0, \ldots, k - 1$. Each of these k components consists of a clique of n vertices labeled $0, \ldots, n - 1$, the intention being that the vertex labeled i represents a choice of variable i being made true in the formula X. Correspondingly in the next $B(r)$ we have columns (cliques) $i = 0, \ldots, n - 1$. The intention is that column i corresponds to the choice of variable i in

the preceding $A(r)$. The idea then is the following. We join the vertex $a[r, i]$ corresponding to variable i, in $A(r)$, to all vertices in $B(r)$ *except* those in column i. This means that the choice of i in $A(r)$ will cover all vertices of $B(r)$ except those in this column. It follows that we *must* choose the dominating element from this column and nowhere else. (There are no connections from column to column.) The columns are meant to be the gap selection that says how many 0's there will be until the next positive choice for a variable. We finally need to ensure that if we chose variable i in $A(r)$ and gap j in column i from $B(r)$ then we need to pick $i + j + 1$ in $A(r + 1)$. This is the role of the gap enforcement component which consists of a set of n vertices (in V_6.) The method is to connect vertex j in column i of $B(r)$ to all of the n vertices $d[r, s]$ *except* to $d[r, i + j + 1]$. The point of this is that if we choose j in column i we will dominate all of the $d[r, s]$ except $d[r, i + j + 1]$. Since we will only connect $d[r, s]$ additionally to $a[r + 1, s]$ and nowhere else, to choose an element of $A[r + 1]$ and still dominate all of the $d[r, s]$ we must actually choose $a[r + 1, i + j + 1]$.

Thus the above provides a selection gadget that chooses k true variables with the gaps representing false ones. We enforce that the selection is consistent with the clauses of X via the clause variables V_3. These are connected in the obvious ways. One connects a choice in $A[r]$ or $B[r]$ corresponding to making a clause C_q true to the variable c_q. Then if we dominate all the clause variables too, we must have either chosen in some $A[r]$ a positive occurrence of a variable in C_q or we must have chosen in $B[r]$ a gap corresponding to a negative occurrence of a variable in C_q, and conversely. We now turn to the formal details.

The vertex set V of G is the union of the following sets of vertices:

$$
\begin{aligned}
V_1 &= \{a[r, s] : 0 \le r \le k - 1, 0 \le s \le n - 1\},\\
V_2 &= \{b[r, s, t] : 0 \le r \le k - 1, 0 \le s \le n - 1, 1 \le t \le n - k + 1\},\\
V_3 &= \{c[j] : 1 \le j \le m\},\\
V_4 &= \{a'[r, u] : 0 \le r \le k - 1, 1 \le u \le 2k + 1\},\\
V_5 &= \{b'[r, u] : 0 \le r \le k - 1, 1 \le u \le 2k + 1\},\\
V_6 &= \{d[r, s] : 0 \le r \le k - 1, 0 \le s \le n - 1\}.
\end{aligned}
$$

For convenience, we introduce the following notation for important subsets of some of the vertex sets above. Let

$$
\begin{aligned}
A(r) &= \{a[r, s] : 0 \le s \le n - 1\},\\
B(r) &= \{b[r, s, t] : 0 \le s \le n - 1, 1 \le t \le n - k + 1\},\\
B(r, s) &= \{b[r, s, t] : 1 \le t \le n - k + 1\}.
\end{aligned}
$$

The edge set E of G is the union of the following sets of edges. In these descriptions we implicitly quantify over all possible indices.

$$
\begin{aligned}
E_1 &= \{c[j]a[r, s] : x_s \in C_j\},\\
E_2 &= \{a[r, s]a[r, s'] : s \ne s'\},\\
E_3 &= \{b[r, s, t]b[r, s, t'] : t \ne t'\},\\
E_4 &= \{a[r, s]b[r, s', t] : s \ne s'\},\\
E_5 &= \{b[r, s, t]d[r, s'] : s' \ne s + t \pmod n\},\\
E_6 &= \{a[r, s]a'[r, u]\},\\
E_7 &= \{b[r, s, t]b'[r, u]\},\\
E_8 &= \{c[j]b[r, s, t] : \exists i\ \overline{x_i} \in C_j, s < i < s + t\},\\
E_9 &= \{d[r, s]a[r', s] : r' = r + 1 \pmod n\}.
\end{aligned}
$$

Suppose X has a satisfying truth assignment τ of weight k, with variables $x_{i_0}, x_{i_1}, \ldots, x_{i_{k-1}}$ assigned the value *true*. Suppose $i_0 < i_2 < \cdots < i_{k-1}$. Let $d_r = i_{r+1(\text{mod } k)} - i_r \pmod n$ for $r = 0, \ldots, k-1$. It is straightforward to verify that the set of $2k$ vertices

$$D = \{a[r, i_r] : 0 \leq r \leq k-1\} \cup \{b[r, i_r, d_r] : 0 \leq r \leq k-1\}$$

is a dominating set in G.

Conversely, suppose D is a dominating set of $2k$ vertices in G. The closed neighborhoods of the $2k$ vertices $a'[0, 1], \ldots, a'[k-1, 1], b'[0, 1], \ldots, b'[k-1, 1]$ are disjoint, so D must consist of exactly $2k$ vertices, one in each of these closed neighborhoods. Also, none of the vertices of $V_4 \cup V_5$ are in D, since if $a'[r, u] \in D$ then necessarily $a'[r, u'] \in D$ for $1 < u' < 2k+1$ (otherwise D fails to be dominating), which contradicts that D contains exactly $2k$ vertices. It follows that D contains exactly one vertex from each of the sets $A(r)$ and $B(r)$ for $0 \leq r \leq k-1$.

The possibilities for D are further constrained by the edges of E_4, E_5, and E_9. The vertices of D in V_1 represent the variables set to *true* in a satisfying truth assignment for X, and the vertices of D in V_2 represent intervals of variables set to *false*. Since there are k variables to be set to *true* there are, considering the indices of the variables mod n, also k intervals of variables to be set to *false*.

The edges of E_4, E_5, and E_9 enforce that the $2k$ vertices in D must represent such a choice consistently. To see how this enforcement works, suppose $a[3, 4] \in D$. This represents that the third of k distinct choices of variables to be given the value *true* is the variable x_4. The edges of E_4 force the unique vertex of D in the set $B(3)$ to belong to the subset $B(3, 4)$. The index of the vertex of D in the subset $B(3, 4)$ represents the difference (mod n) between the indices of the third and fourth choices of a variable to receive the value *true*, and thus the vertex represents a range of variables to receive the value *false*. The edges of E_5 and E_9 enforce that the index t of the vertex of D in the subset $B(3, 4)$ represents the "distance" to the next variable to be set *true*, as it is represented by the unique vertex of D in the set $A(4)$.

It remains only to check that the fact that D is a dominating set ensures that the truth assignment represented by D satisfies X. This follows by the definition of the edge sets E_1 and E_8. □

Because DOMINATING SET can be easily reduced to WEIGHTED CNF SATISFIABILITY with no negated literals, the above theorem shows the surprising fact that WEIGHTED SATISFIABILITY reduces to MONOTONE WEIGHTED CNF SATISFIABILITY. (The reduction is straightforward. Let $\{x_1, \ldots, x_n\}$ denote the set of vertices of the given graph G, and we will interpret them as input variables for our circuit $C(G)$. Have a layer of Or gates directly below the variables. These are also one per input variable and we will label them $g_1, \ldots, g_n$. Make x_i and input to the Or gate g_j precisely in the case that (x_i, x_j) is an edge of G. Now to complete the circuit, we have one large *And* gate with inputs from each of the Or gates. It is easy to see that satisfying assignments correspond directly to dominating sets and conversely.) Interpreted in terms of circuits, this combinatorial reduction plays a crucial role in the fundamental completeness results surveyed in the next section. We can use this reduction in the main theorem (Theorem 3.7) because the reduction proves rather more than is stated in Theorem 2.1 and the relavant properties we need are stated in the remark below. (It is also used to generalize Theorem 2.1 to MONOTONE $2t$-NORMALIZED BOOLEAN FORMULAE in Corollary 3.8.)

Remark. An important fact about the above proof is the following. For our fixed k the enforcement gadgetry causes us to choose the $2k$ vertices, k from the $A(r)$'s and k from the B's, and there is a 1–1 correspondence between weight k assignments to X and size $2k$ sets that *dominate the graph G' which denotes the gadget, that is, G without the clause*

connections and vertices. Define a *weak dominating set* to be a set of $(2k)$ vertices that dominates the gadget of G. The fact we will use in the proof of the normalization theorem is that under the 1–1 correspondence between weight k assignments to X and weak dominating sets, if a clauses $C_{i_1}, \ldots, C_{i_p}$ are the subset of the set of clauses satisfied by some weight k assignment, then $c_{i_1}, \ldots, c_{i_p}$ are exactly the clause vertices dominated by the corresponding size $2k$ weak dominating set in G. That is, not only is there a correspondence between weight k satisfying assignments for X and weight $2k$ dominating sets in G, but in fact there is an exact correspondence between *all* weight k assignments together with the clauses they satisfy, and with *all* weak dominating sets and the clause vertices they dominate.

The DOMINATING SET reduction also allows for a number of other applications. For instance, consider the problem below.

WEIGHTED {0, 1}-INTEGER PROGRAMMING
Instance: A binary matrix A and a binary vector $\mathbf{b}$.
Parameter: k.
Question: Does $A \cdot \mathbf{x} \geq \mathbf{b}$ have a binary solution of weight k?

We have the following corollary.

COROLLARY 2.2. *WEIGHTED CNF SATISFIABILITY reduces to WEIGHTED* {0, 1}*-INTEGER PROGRAMMING.*

Proof. Let (X, k) be an instance of MONOTONE WEIGHTED CNF SAT. Let $C_1, \ldots, C_p$ list the clauses of X and $x_1, \ldots, x_m$ list the variables. Let A be the matrix $\{a_{i,j} : i = 1, \ldots, p, j = 1, \ldots, m\}$ with $a_{i,j} = 1$ if x_j is present in C_i and $a_{i,j} = 0$ otherwise. Let $\mathbf{b}$ be the vector with 1 in the jth position for $j = 1, \ldots, p$. It is easy to see that $A \cdot \mathbf{x} \geq \mathbf{b}$ has a solution of weight k iff X has a satisfying assignment of weight k (and the reasoning is reversible). □

In passing we remark that the situation of Corollary 2.2, where a classical reduction can easily be modified for a normal parameterized case, is exceedingly rare. It seems to be that it is almost *never* the case that a classical reduction gives rise to a parameterized one.

3. A completeness theory for fixed-parameter intractability. In order to frame a completeness theory to address the apparent fixed-parameter intractability of DOMINATING SET and other problems, we need to define appropriate classes of parameterized problems. As one might expect, satisfiability occupies a central role in our investigations. But now the situation is apparently much more complex than in the classical case. For instance, while classically there is no difference between CNF SATISFIABILITY and SATISFIABILITY for general formulae in propositional logic, there seems to be *no* parameterized reduction computing general SATISFIABILITY from CNF SATISFIABILITY and hence they do seem to be genuinely of different complexity from a parameterized point of view. Considerations such as these lead to the conclusion that there seem to be *many* different parameterized degree classes of natural problems. This is quite different from the situation in classical, say, *NP*-completeness results where virtually all natural problems which are not in P seem to be *NP* complete.

Current measure of intractability. At the present time if we wish to prove parameterized intractability, we show that the problem at hand can compute WEIGHTED 3CNF SATISFIABILITY. This is the class we call $W[1]$ and as we said before, we seem to have very strong evidence that it is intractable. We refer the reader to Downey and Fellows [56].

t-normalized formulae. In fact, there does not seem to be any reduction from parameterized CNF to 3CNF. This leads naturally to the realization that the logical depth of a propositional formula affects its parameterized complexity. Thus if CNF is thought of as products-of-sums of literals, then we can define a propositional formula to be *t-normalized* if it is of the form products-of-sums-of-products ... of literals with t-alternations. (Hence

2-normalized is the same as CNF.) Now we believe that for all t, WEIGHTED SATISFIABILITY for t-normalized formulae is strictly easier than WEIGHTED SATISFIABILITY for $t+1$-normalized formulae. This suggests a hierarchy based on the complexity of t-normalized formulae. It turns out that this hierarchy is quite useful for the classification of the complexity of many natural problems, and t-normalized formulae do give quite a bit more computational power. We make this more precise through the introduction of the circuit-based classes below.

The classes that we define below are intuitively based on the complexity of the circuits required to check a solution. The size of a circuit is as usual the number of gates in the circuit.

We first define decision circuits in which some gates have bounded fan-in and some have unrestricted fan-in. It is assumed that fan-out is never restricted.

DEFINITION 3.1. *A circuit is of* mixed type *if it consists of circuits having gates of the following kinds:*

(1) Small gates*:* not *gates,* and *gates and* or *gates with bounded fan-in. We will usually assume that the bound on fan-in is* 2 *for* and *gates and* or *gates, and* 1 *for* not *gates.*

(2) Large gates*:* And *gates and* Or *gates with unrestricted fan-in.*

We will use lowercase to denote small gates (*or* gates and *and* gates), and uppercase to denote large gates (*Or* gates and *And* gates).

DEFINITION 3.2 (depth and weft). *The* depth *of a circuit C is defined to be the maximum number of gates (small or large) on an input–output path in C. The* weft *of a circuit C is the maximum number of large gates on an input–output path in C.*

DEFINITION 3.3. *We say that a family of circuits $\mathcal{F}$ has* bounded depth *if there is a constant h such that every circuit in the family $\mathcal{F}$ has depth at most h. We say that $\mathcal{F}$ has* bounded weft *if there is constant t such that every circuit in the family $\mathcal{F}$ has weft at most t. $\mathcal{F}$ is* monotone *if the circuits of $\mathcal{F}$ do not have not-gates. A circuit C is a* decision circuit *if it has a single output. A decision circuit C* accepts *an input vector x if the single output gate has value* 1 *on input x.*

DEFINITION 3.4. *Let $\mathcal{F}$ be a family of decision circuits. We allow that $\mathcal{F}$ may have many different circuits with a given number of inputs. To $\mathcal{F}$ we associate the parameterized circuit problem $L_{\mathcal{F}} = \{(C, k) : C \in \mathcal{F}$ accepts an input vector of weight $k\}$.*

DEFINITION 3.5. *A parameterized problem L belongs to $W[t]$* (monotone $W[t]$) *if L uniformly (m-)reduces to the parameterized circuit problem $L_{\mathcal{F}}$ for the family $\mathcal{F}_{h,t}$ of depth h, mixed type (monotone) decision circuits of weft at most t.*

DEFINITION 3.6. *We denote the class of fixed-parameter tractable problems as FPT.*

Thus we have the containments

$$FPT \subseteq W[1] \subseteq W[2] \subseteq \cdots$$

and we conjecture that each of these containments is proper. We term the union of these classes together with two other classes $W[SAT] \subseteq W[P]$, the W-Hierarchy. Here $W[P]$ denotes the class obtained by having no restriction on depth, i.e., P-size circuits, and $W[SAT]$ denotes the restriction to boolean circuits of P-size. We do not explore $W[SAT]$ or $W[P]$ here but do so in Abrahamson, Downey, and Fellows [1], [2].

Our main result shows that WEIGHTED CNF SATISFIABILITY is complete for $W[2]$ and that similar problems are complete for each level of the W-Hierarchy of parameterized problem classes. This theorem, Theorem 3.7, plays a role in our theory analogous to Cook's theorem for NP-completeness, in the following sense. Usually proofs of membership in a particular $W[t]$ class are easy, so the circuit definition is easy to reduce *to*, whereas the t-normalized formulae provide problems that are easy to reduce *from* to establish hardness. The other view of the Cook(–Levin) theorem is that it connects a generic problem from Turing machines to SATISFIABILITY and hence SATISFIABILITY is very unlikely to be tractable.

This other view of the Cook(–Levin) theorem is not pursued here but is established for $W[1]$ in Downey and Fellows [56], as we mentioned earlier. The present paper, and particularly Theorem 3.7, provide an important technical framework for hardness results.

It is interesting that the combinatorial reduction of Theorem 2.1, and the subsequent remark play a key role (as a "change of variables") in our proof of Theorem 3.7. Thus the entire argument that DOMINATING SET is complete for $W[2]$ actually uses this combinatorial reduction *twice*. Recall the notion of t-normalized formula we discussed earlier. It naturally gives rise to the following problem.

WEIGHTED t-NORMALIZED SATISFIABILITY
Input: A t-normalized boolean expression X.
Parameter: A positive integer k.
Question: Does X have a satisfying truth assignment of weight k?

THEOREM 3.7. *For $t \geq 2$ WEIGHTED t-NORMALIZED SATISFIABILITY is many:1 complete for $W[t]$.*

Proof. Let $L \in W[t]$. Let $\mathcal{F}$ be the family of circuits of depth bounded by h and weft bounded by t to which L reduces. It suffices to reduce $L_{\mathcal{F}}$ to WEIGHTED t-NORMALIZED SATISFIABILITY. An instance of the latter problem may be viewed as a pair consisting of a positive integer k and a circuit having t alternating layers of *And* and *Or* gates corresponding to the t-normalized expression structure P-o-S-o-P- ..., and having a single output *And* gate. Thus the argument essentially shows how to "normalize" the circuits in $\mathcal{F}$.

Let (C, k) be an instance of $L_{\mathcal{F}}$. We show how to determine whether C accepts a weight k input vector by consulting an oracle for WEIGHTED t-NORMALIZED SATISFIABILITY (viewed as a problem about circuits) for finitely many weights k'. The algorithm for this determination will be uniform in k, and run in time $f(k)n^{\alpha}$ where n is the size of the circuit C. The exponent α will be a (possibly exponential) function of h and t. This is permissible, since every circuit in $\mathcal{F}$ observes these bounds on depth and weft.

Step 1. *The reduction to tree circuits.*

The first step is to transform C into a *tree circuit* C' (or $formula$) of depth and weft bounded by h and t, respectively. In a tree circuit every logic gate has fan-out one. (The input nodes may have large fan-out.) The transformation is accomplished by replicating the portion of the circuit above a gate as many times as the fan-out of the gate, beginning with the top level of logic gates and proceeding downward level by level. (We regard a decision circuit as arranged with the inputs on top and the output on the bottom.) The creation of C' from C may require time $O(n^{O(h)})$ and involve a similar blow-up in the size of the circuit. The tree circuit C' accepts a weight k input vector if and only if the original circuit C accepts a weight k input vector.

Step 2. *Moving the not gates to the top of the circuit.*

Let C denote the circuit we receive from the previous step (we will use this notational convention throughout the proof). Transform C into an equivalent circuit C' by commuting the *not* gates to the top, using DeMorgan's laws. This may increase the size of the circuit by at most a constant factor. The tree circuit C' thus consists (from the top) of the input nodes, with *not* gates on some of the lines fanning out from the inputs. In counting levels we consider all of this as level 0, and may refer to negated fan-out lines from the input nodes as negated inputs. Next, there are levels consisting only of large and small *and* and *or* gates, with a single output gate (which may be of either principal logical denomination at this point).

Step 3. *Homogenizing the layers.*

The goal of this step is to reduce to the situation where all of the large gates are at the bottom of the circuit, in alternating layers of large *And* and *Or* gates. To achieve this we work from the bottom up, with the first task being to arrange for the output gate to be large.

Let C denote the circuit received from the previous step. Suppose the output gate z is small. Let $C[z]$ denote the connected component of C including z that is induced by the set of small gates. Thus all gates providing input to $C[z]$ are either large or are input gates of C. Because of the bound h on the depth of C, there are at most 2^h inputs to $C[z]$. The function of these inputs computed by $C[z]$ is equivalent to a product-of-sums expression E_z having at most 2^{2^h} sums, with each sum a product of at most 2^h inputs. Let C' denote the circuit equivalent to C obtained by replacing the small gate output component $C[z]$ with E_z, duplicating subcircuits of C as needed to provide the inputs to the depth 2 circuit representing E_z. (The "product" gate of E_z is now the output gate of C'.) This entails a blow-up in size by a factor bounded by 2^{2^h}. Since h is an absolutely fixed constant (not dependent on n or k) this blow-up is "linear" and permitted. Note that E_z and therefore C' are easily computed in a similar amount of time to this size blow-up.

Let p denote the output *and* gate of C' (corresponding to the product in E_z). Let $s_1, \ldots, s_m$ denote the *or* gates of C' corresponding to the sums of E_z. We consider all of these gates to be *small*, since the number of inputs to them does not depend on n or k. (Equivalently, if the gates of these two levels were replaced by binary input gates, we would see that the reduction of C to C' has increased circuit depth from h to 2^h.)

Each *or* gate s_i of C' has three kinds of input lines: those coming from large *And* gates, those coming from large *Or* gates, and those coming from input gates of C'. We will use the same symbol to denote an input line, the subcircuit of C' that computes the value on that line, or the Boolean expression corresponding to the subcircuit (since C' is a tree circuit, it is equivalent to a Boolean expression). Let these three groups of inputs be denoted, respectively, by

$$S_{i,\wedge} = \{s_i[\wedge, j] : \ j = 1, \ldots, m_{i,\wedge}\},$$

$$S_{i,\vee} = \{s_i[\vee, j] : \ j = 1, \ldots, m_{i,\vee}\},$$

$$S_{i,\top} = \{s_i[\top, j] : \ j = 1, \ldots, m_{i,\top}\},$$

and define

$$S_i = S_{i,\wedge} \cup S_{i,\vee} \cup S_{i,\top}.$$

For each line $s_i[\vee, j]$ of C' coming from a large *Or* gate u, let

$$S_{i,\vee,j} = \{s_i[\vee, j, k] : \ k = 1, \ldots, m_{i,\vee,j}\}$$

denote the set of input lines to u in C'. Similarly, for each line $s_i[\wedge, j]$ of C' coming from a large *And* gate v, let

$$S_{i,\wedge,j} = \{s_i[\wedge, j, k] : \ k = 1, \ldots, m_{i,\wedge,j}\}$$

denote the set of input lines to v in C'.

Let

$$k' = \sum_{i=1}^{m} (1 + m_{i,\vee}).$$

The integer k' is the number of *or* gates (counting both large and small gates) that are either part of $C[z]$ or directly supply input to $C[z]$. Note that k' is bounded above by $2^h \cdot 2^{2^h}$.

We describe how to produce a weft t circuit C'' from C' that accepts an input vector of weight $k'' = k + k'$ if and only if C' (and therefore C) accepts an input vector of weight k. The tree circuit C'' will have a large *And* gate giving the output.

Let $x_1, \ldots, x_n$ denote the inputs to C'. The circuit C'' has additional input variables that, for the most part, correspond to the input lines to the *or* gates singled out for attention above. The set V of new input variables is the union of the following groups of variables:

$$V = \left(\bigcup_{i=1}^{m} V_i\right) \cup \left(\bigcup_{i=1}^{m}\bigcup_{j=1}^{m_{i,\vee}} V_{i,j}\right),$$

where

$$V_i = \{v_i[\wedge, j] : 1 \le j \le m_{i,\wedge}\} \cup \{v_i[\vee, j] : 1 \le j \le m_{i,\vee}\} \cup \{v_i[\top, j] : 1 \le j \le m_{i,\top}\}$$

and

$$V_{i,j} = \{v_i[\vee, j, k] : 1 \le k \le m_{i,\vee,j}\} \cup \{n[i, j]\}.$$

The circuit C'' is represented by the Boolean expression

$$C'' = E_1 \cdot E_2 \cdot E_3 \cdot E_4 \cdot E_5 \cdot E_6 \cdot E_7,$$

where

$$E_1 = \prod_{i=1}^{m} \left(\sum_{u \in V_i} u\right),$$

$$E_2 = \prod_{i=1}^{m} \prod_{u \ne v,\ u,v \in V_i} (\neg u + \neg v),$$

$$E_3 = \prod_{i=1}^{m}\prod_{j=1}^{m_{i,\vee}} \left(\sum_{u \in V_{i,j}} u\right),$$

$$E_4 = \prod_{i=1}^{m}\prod_{j=1}^{m_{i,\vee}} \prod_{u \ne v}^{u,v \in V_{i,j}} (\neg u + \neg v),$$

$$E_5 = \prod_{i=1}^{m}\prod_{j=1}^{m_{i,\wedge}}\prod_{k=1}^{m_{i,\wedge,j}} (s_i[\wedge, j, k] + \neg v_i[\wedge, j]),$$

$$E_6 = \prod_{i=1}^{m}\prod_{j=1}^{m_{i,\vee}} (\neg v_i[\vee, j] + \neg n[i, j]),$$

$$E_7 = \prod_{i=1}^{m}\prod_{j=1}^{m_{i,\vee}}\prod_{k=1}^{m_{i,\vee,j}} (s_i[\vee, j, k] + \neg v_i[\vee, j, k]).$$

The size of C'' is bounded by $|C'|^2$.

Claim 1. The circuit C'' has weft t.

To see this, note first that since $t \geq 2$, any input–output path beginning from a new input variable (in V) that has at most two large gates as the expression for C'' is essentially a product-of-sums. In E_5 and E_7 the sums involve subexpressions of C'; any input–output path from an original input variable (of C') passes through one of these *or* gates. Observe that in C' these subexpressions have weft at most $t - 1$. The sums of E_5 and E_7 are small, so these paths further encounter only the bottommost large *And* gate.

Claim 2. The circuit C'' accepts an input vector of weight k'' if and only if C' accepts an input vector of weight k.

First note that any input vector of weight k'' accepted by C'' must set exactly one variable in each of the sets of variables V_i (for $i = 1, \ldots, m$) and $V_{i,j}$ (for $i = 1, \ldots, m$ and $j = 1, \ldots, m_{i,\vee}$) to 1 and all of the others in the set to 0 in order to satisfy $E_1 \cdot E_2 \cdot E_3 \cdot E_4$. It follows that any such accepted input must set exactly k of the original variables of C' to 1, by the definition of k''.

The role of the (new) variables set to 1 in the sets of variables that represent inputs to *or* gates is to indicate an accepting computation of C' on the weight k input of old variables. The expressions $E_5, \ldots, E_7$ enforce the correctness of this representation in C'' of the computation of C'.

The expression E_5 ensures that if the new variable $v_i[\wedge, j]$ is set to 1, indicating that the subexpression $s_i[\wedge, j]$ of C' evaluates to 1, then every argument $s_i[\wedge, j, k]$ must evaluate to 1. (Note that subexpressions $s_i[\wedge, j, k]$ appear in C'' while the subexpressions $s_i[\wedge, j]$ do not. The computations performed in C' by the latter are simply represented by the values of the input variables in V.) The role of the variables $n[i, j]$ is to represent that "none of the inputs" to the *or* gate has the value 1. The expression E_6 enforces that if this situation is represented, then the output of the gate is not represented as having the value 1. The expression E_7 ensures that if the new variable $v_i[\vee, j, k]$ has the value 1, indicating that the subexpression $s_i[\vee, j, k]$ of C' evaluates to 1, then this subexpression must in fact evaluate to 1.

By the above, we may now assume that the circuit we are working with has a large output gate (which may be of either denomination). Renaming for convenience, let C denote the circuit we are working with under this assumption.

If g and g' are gates in C of the same logical character ($\wedge$ or $\vee$) with the output of g going to g', then they can be consolidated into a single gate without increasing weft if g is small and g' is large. We term this a *permitted contraction*. Note that if g is large and g' is small then contraction may not preserve weft. We will assume that permitted contractions are performed whenever possible, interleaved with the following two operations.

(1) *Replacement of bottommost small gate components.*

As at the beginning of Step 3, let $C_1, \ldots, C_m$ denote the bottommost connected components of C induced by the set of small gates and having at least one large gate input. Since the output gate of C is large, each C_i gives output to a large gate g_i. If g_i is an *And* gate, then C_i should be replaced with product-of-sums circuitry equivalent to C_i. If g_i is an *Or* gate, then C_i should be replaced with equivalent sum-of-products circuitry. Note that in either case this immediately creates the opportunity for a permitted contraction. As per the discussion at the beginning of Step 3, this replacement circuitry is small, and this operation may increase the size of the circuit by a factor of 2^{2^h}. This step will be repeated at most h times, as we are working from the bottom up in transforming C.

(2) *Commuting small gates upward.*

After (1), and after the permitted contractions, the bottommost small gate components are each represented in the modified circuit C' by a single small gate h_i giving output to g_i.

Without loss of generality, all of the arguments to h_i may be considered to come from large gates. (The only other possibility is that an input argument to h_i may be from the input level of the circuit, but there is no increase in weft in treating this as an honorary large gate for convenience.) Suppose that g_i is an *And* gate and that h_i is an *or* gate (the other possibility is handled dually).

There are three possible cases:

(i) All of the arguments to h_i are from *Or* gates.

(ii) All of the arguments to h_i are from *And* gates.

(iii) The arguments to h_i include both large *Or* gates and large *And* gates.

In case (i), we may consolidate h_i and all of the gates giving input to h_i into a single large *Or* gate without increasing weft.

In case (ii), we replace the small $\vee$ (h_i) of large $\wedge$'s with the equivalent (by distribution) large $\wedge$ of small $\vee$'s. Since h_i may have 2^h inputs, this may entail a blow-up in the size of the circuit from n to n^{2^h}. This does not increase weft, and creates the opportunity for a permitted contraction.

In case (iii), we similarly replace h_i and its argument gates with circuitry representing a product-of-sums of the inputs to the arguments of h_i. The difference is that in this case, the replacement is a large $\wedge$ of large (rather than small) $\vee$ gates. Weft is preserved when we take advantage of the contraction now permitted between the large $\wedge$ gate and g_i.

We may achieve our purpose in this step by repeating the cycle of (1) and (2). At most h repetitions are required. The total blow-up in the size of the circuit in this step is crudely bounded by $n^{2^{h^2}}$.

Step 4. *Removing a bottommost Or gate.*

By a Turing reduction, we can determine whether a tree circuit giving output from an *Or* gate accepts a weight k input vector by simply making the same determination for each of the input branches (subformulae) to the gate.

In order to accomplish this step by a many:1 reduction, we do the following. Let b be the number of branches of the circuit C with bottommost *Or* gate that we receive at the beginning of this step. We modify C by creating new inputs $x[1], \ldots, x[b]$. The purpose of these input variables is to indicate which branch of C accepts a weight k input vector. Let $C_1, \ldots, C_b$ be the branches of C, so that C is represented by the expression $C_1 + \cdots + C_b$. The modified circuit C' is represented by the expression

$$C' = (x[1] + \cdots + x[b]) \cdot \prod_{1 \le i < j \le b} (\neg x[i] + \neg x[j]) \cdot \prod_{1 \le i \le b} (C_i + \neg x[i]).$$

The first two product terms of the above expression ensure that exactly one of the new variables must have value 1 in an accepted input vector. The modified circuit C' accepts a weight $k + 1$ input vector if and only if C accepts a weight k input vector. For weft at least two, the transformation is weft-preserving and yields a circuit C' with bottommost *And* gate, but possibly with *not* gates at the lower levels. Thus it may be necessary to repeat Steps 2 and 3 to obtain a homogenized circuit with bottommost *And* gate.

Step 5. *Organizing the small gates.*

The tree circuit C received from the previous step has the following properties: (i) the output gate is an *And* gate, (ii) from the bottom, the circuit consists of layers which alternately consist of only *And* gates or only *Or* gates, for up to t layers, and (iii) above this, there are branches B of height $h' = h - t$ consisting only of small gates. Since a small gate branch B has bounded depth, it has at most $2^{h'}$ gates, and thus in constant time (since h is fixed), we can find either (1) an equivalent sum-of-products circuit with which to replace B, as required

by Case 1 of Step 6 below, or (2) an equivalent product-of-sums circuit, as required by Case 2 of Step 6.

In this step, all such small gate branches B of C are replaced in this way, appropriately for the relevant case of Step 6. In Case 1, the depth 2 sum-of-products circuits replacing the small gate branches B have a bottomost *or* gate g_B of fan-in at most $2^{2^{h'}}$, and the *and* gates feeding into g_B have fan-in at most $2^{h'}$, so the weft of the circuit has been preserved by this transformation, which may increase the size of C by the constant factor $2^{2^{h'}}$. The topmost level of large gates (to which the branches B are attached in C) consists of *Or* gates, in Case 1, so that the gates g_B can be merged into this topmost level. Merging is performed similarly in Case 2, where the replacing circuits are products-of-sums, and the topmost level of large gates consists of *And* gates. For the next step we consider two cases, depending on whether the topmost level of large gates consists of *Or* gates or *And* gates. (Essentially, this corresponds to whether weft t is even or odd.)

Step 6. *A monotone change of variables (two cases).*

In this step (in both cases) we employ a "change of variables" based on the combinatorial reduction of Theorem 2.1. The goal is to obtain an equivalent circuit that has the property that either all the inputs are MONOTONE (Case 1) (i.e., no inverters in the circuit), or all the inputs are negated with no other inverters in the circuit, which we call ANTIMONOTONE. (Actually in this case we will have *some* of the inputs positive but these will only be *enforcers* as we will see. So we should call this case NEARLY ANTIMONOTONE) (Case 2). The point of this step becomes apparent in the next step when we use the special character of the circuit thus constructed to enable us to eliminate the small gates.

Consider the reduction of Theorem 2.1, especially in the light of the remark following the proof. This reduction consists of two parts. The first is the ring of selection gadgets which allow variable choice, gap choice, and then gap enforcement; the second part is consistency obtained by clause wiring. The idea is to "hard wire" the selection and consistency parts of the construction into the circuit, the point being that we can replace positive instances of variable fan-out in the original circuit by outputs corresponding to choice of that variable in the positive selection component. We can replace negative fan-out in the original circuit by the appropriate sets of gap variables. Finally we can wire in the fact that we need a dominating set and other enforcements by using the facts that we will only look at a weight $2k$ input, and an *And* of *Or*'s, which will not add to the *weft* of the circuit. We argue more precisely below, and also prove in two parts that the whole process cn be accomplished without increasing *weft*, given that the weft is ≥ 2.

Suppose the inputs to the circuit C received at the beginning of this step are $x[1], \ldots, x[n]$, and suppose that the output gate of C is an *And* gate. Let Y denote the boolean expression having $2n$ clauses, with each clause consisting of a single literal, and with one clause for each of the $2n$ literals of the n input variables. The reduction of Theorem 2.1 allows us to translate Y into a monotone formula via dominating set, thus capturing monotonically all the relevant input settings. Thus, let G_Y be the graph constructed for this expression as in the proof of Theorem 2.1. Note that only part of G_Y will actually be wired into C.

Keeping this in mind, and using the variable (vertex) set obtained from G_Y, the change of variables is implemented for C as follows. (1) Create a new input for each vertex of G_Y that is not a clause vertex. (2) Replace each positive input fan-out of $x[i]$ in C with an *Or* gate having k new input variable arguments corresponding to the vertices to which the clause vertex for the clause $(x[i])$ of Y is adjacent in G_Y. (3) Replace each negated fan-out line of $x[i]$ with an *Or* gate having $O(n^2)$ new input variable arguments corresponding to the vertices to which the clause vertex for the clause $(\neg x[i])$ of Y is adjacent in G_Y. (4) Merge with the output *And* gate of C a new circuit branch corresponding to the product-of-sums expression,

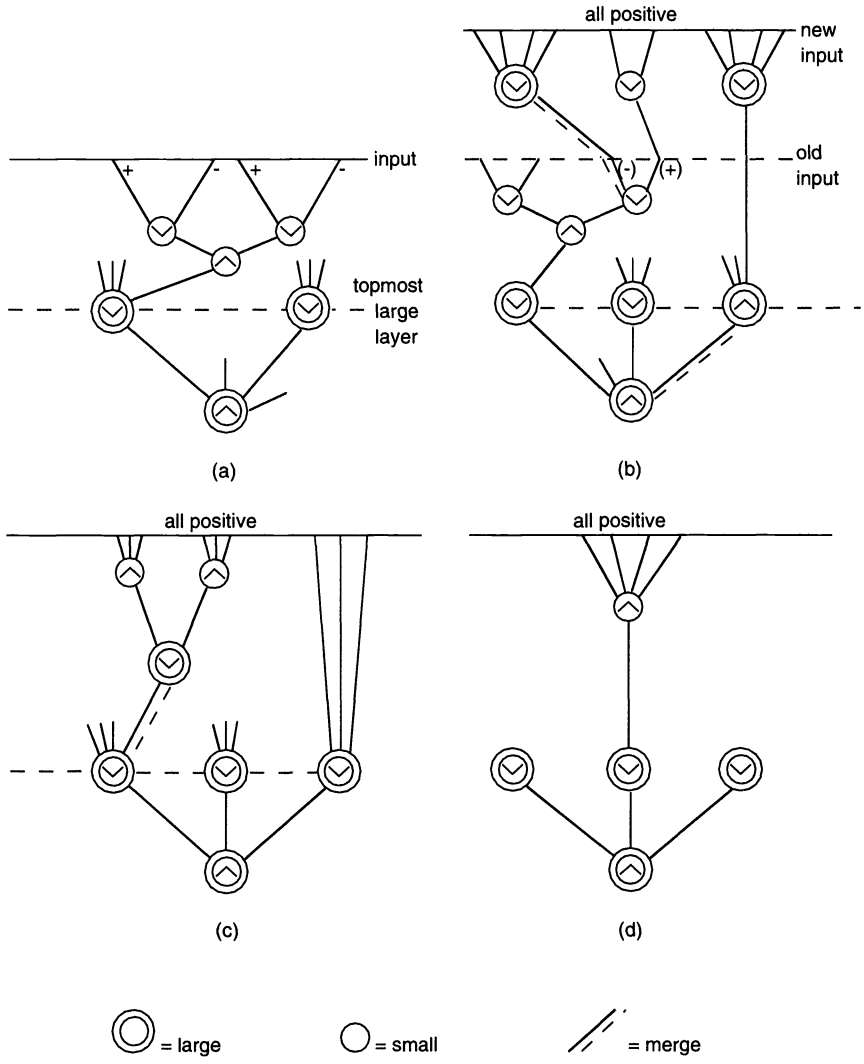


FIG. 2. *A topmost layer of large Or gates.*

where the product is taken over all nonclause vertices of G_Y, and the sum for a vertex u is the sum of the new inputs corresponding to the nonclause vertices in $N[u]$ (this is the dominating set and other enforcements).

The modified circuit C' obtained in this way accepts a weight $2k$ input vector if and only if the original circuit C accepts a weight k input vector. The proof of this is essentially the same as for Theorem 2.1. If all of the *not* gates of C are at the top, then the circuit C' will be MONOTONE. However, to see that this change of variables can be employed to obtain a monotone or nearly antimonotone circuit *without* increasing weft, we must consider two cases.

Case 1. *The topmost large-gate level consists of Or gates.*

Let C denote the circuit obtained from Step 5 and perform a change of variables as described above. The sequence of transformations of C for this step is shown schematically in Fig. 2.

The result is a circuit C' with no *not* gates. The input weight we are now concerned with is $2k$, and the construction of C' from C may involve quadratic blow-up.

Next, we move the small *and* gates on the second level upward past the *Or* gates introduced by the change of variables, and then merge the *Or* gates down to the topmost large layer (of *Or* gates).

Case 2. *The topmost large-gate level consists of And gates.*

Here we use a similar argument, beginning with a trick. Below each gate of the topmost large-gate level (of *And* gates), a double negation is introduced (equivalently). One of the *not* gates is moved to the top of the circuit (by DeMorgan's identities). This is followed by a change of variables based on Theorem 2.1, as in Case 1. The second level *and* gates are commuted upwards, and the *Or* gates of the second and third levels are merged, as in Case 1. So now the circuit has no negated inputs and no inverters except the residual ones below the top layer of *Or* gates. Finally, these remaining *not* gates are commuted to the top. Note that this means that all fanouts are negated except the ones to the enforcement *Or* gate added during Step 4.

We are now in position for the last step.

Step 7. *Eliminating the remaining small gates.*

If we regard the inputs to C as variables, this step consists of another "change of variables." Let k be the relevant weight parameter value supplied by the last transformation. In this step we will produce a circuit C' corresponding directly to a t-normalized boolean expression (that is, consisting only of t alternating layers of *And* and *Or* gates) such that C accepts a weight k input vector if and only if C' accepts a vector of weight $k' = k \cdot 2^{k+1} + 2^k$.

Suppose that C has m remaining small gates. In Case 1, these are *and* gates and the inputs are all positive. In Case 2, these are *or* gates and the inputs to these gates are all negated. For $i = 1, \ldots, m$ we define the sets A_i of the inputs to C to be the sets of input variables to these small gates. The central idea for this step is to create new inputs representing the sets A_i of inputs to C.

For example, suppose (Case 1) that the output of the small *and* gate g_i in C is the boolean product $(abcd)$ of the inputs a, b, c, d to C. Thus $A_i = \{a, b, c, d\}$. The gate g_i can be eliminated by replacing it with an input line from a new variable $v[i]$ which represents the predicate $a = b = c = d = 1$. (This representation, of course, will need to be enforced by additional circuit structure.) Similarly (Case 2) if g_i computes the value $(\bar{a} + \bar{b} + \bar{c} + \bar{d})$ then g_i can be replaced by a negated input line from $v[i]$.

Let $x[j]$ for $j = 1, \ldots, s$ be the input variables to C. We introduce new input variables of the following kinds:

(1) One new variable $v[i]$ for each set A_i for $i = 1, \ldots, m$ to be used as indicated above.

(2) For each $x[j]$ we introduce 2^{k+1} copies $x[j, 0], x[j, 1], x[j, 2], \ldots, x[j, 2^{k+1} - 1]$.

(3) "Padding" consisting of 2^k meaningless variables (inputs not supplying output to any gates) $z[1], \ldots, z[2^k]$.

We add to the circuit an enforcement mechanism for the change of variables. The necessary requirements can be easily expressed in P-o-S form, and thus can be incorporated into the bottom two levels of the circuit as additional *Or* gates attached to the bottommost (output) *And* gate of the circuit.

We require the following implications concerning the new variables:

(1) The $s \cdot 2^{k+1}$ implications, for $j = 1, \ldots, s$ and $r = 0, \ldots, 2^{k+1} - 1$,

$$x[j, r] \Rightarrow x[j, r + 1 \pmod{2^{k+1}}].$$

(2) For each containment $A_i \subseteq A_{i'}$, the implication

$$v[i'] \Rightarrow v[i].$$

(3) For each membership $x[j] \in A_i$, the implication

$$v[i] \Rightarrow x[j, 0].$$

(4) For $i = 1, \ldots, m$ the implication

$$\left(\prod_{x[j] \in A_i} x[j, 0] \right) \Rightarrow v[i].$$

It may be seen that this transformation increases the size of the circuit by a linear factor exponential in k. We make the following argument for the correctness of the transformation.

If C accepts a weight k input vector, then setting the corresponding copies $x[i, j]$ among the new input variables accordingly, together with appropriate settings for the new "collective" variables $v[i]$ yields a vector of weight between $k \cdot 2^{k+1}$ and $k \cdot 2^{k+1} + 2^k$ that is accepted by C'. The reason the weight of this corresponding vector may fall short of $k' = k \cdot 2^{k+1} + 2^k$ is that not all of the subsets of the k input variables to C having value 1 may occur among the sets A_i. An accepted vector of weight exactly k' can be obtained by employing some of the "padding" input variables $z[i]$ to C'

Note that the seemingly simpler strategy of creating a new input variable for each set of at most k inputs to C would not serve our purposes, since it would involve increasing the size n of the circuit to possibly n^k. (We are limited in our computational resources for the reduction to $f(k)n^\alpha$. The constant α can be an arbitrary function of the depth and weft bounds h and t, but not k.)

For the other direction, suppose C' accepts a vector of weight k'. Because of the implications in (1) above, exactly k sets of copies of inputs to C must have value 1 in the accepted input vector. Because of implications (2)–(4), the variables $v[i]$ must have values in the accepted input vector compatible with the values of the sets of copies. By the construction of C', this implies there is a weight k input vector accepted by C. □

COROLLARY 3.8. (i) *For $t > 0$, MONOTONE $W[2t] = W[2t]$.*

(ii) *For $t > 0$ WEIGHTED MONOTONE $2t$-NORMALIZED SATISFIABILITY is $W[2t]$-complete.*

(iii) *For $t > 0$, MONOTONE $2t + 1$-NORMALIZED SATISFIABILITY is in $W[2t]$.*

Proof. (i) The proof come from the analysis of Step 6, Case 1.

(ii) After Step 7, we can apply Step 6, Case 1 (again) to a $2t$-normalized formula. The result is a $2t$ normalized monotone formula. (This time all small gates go away by gluing the DOMINATING SET reduction, or Theorem 2.1.)

(iii) This result follows by the transformations of Step 7, applied to the a $2t+1$-normalized monotone formula. □

We remark that Corollary 3.8 leads one to conjecture that MONOTONE $W[2t + 1] = W[2t]$.

We also remark that in Downey and Fellows [56] we have proven the complementary result for $W[2t+1]$ by showing that $W[2t+1]$ contains ANTIMONOTONE $2t+2$-NORMALIZED SATISFIABILITY. It is an open question whether ANTIMONOTONE CNFSAT is in $W[1]$. The problem is that the relevant gadgets seem to need two levels to enact. Note that the above theorem fails to identify a problem complete for $W[1]$. In [56] we will also show that INDEPENDENT SET and a number of other natural parameterized problems are complete for $W[1]$. There we show that $W[1] = W[1, 2]$ where $W[1, 2]$ is equivalent to the problem of, given a formula X in conjunctive normal form and of clause size two, does X have a satisfying assignment of weight k? While the unparameterized problem that considers (X, k) is *NP*-complete (easy reduction from INDEPENDENT SET), variations of this are classically

P-time or FPT. For instance the problem that asks if there is any satisfying assignment is well known to be P-time, and the problem that asks if there is a satisfying assignment of weight *less than or equal to* k is FPT as follows. Let X be the $2 - CNF$ formula and let k be given. If X has no clause without negated literals, we are done since the assignment with all false works. Otherwise choose some clause C with only positive literals. One of these must be made true so we begin a tree of possibilities. Continue inductively in this way. As the clause size is bounded here by 2 this only gives a factor of 2^k.

The $W[t]$ hierarchy reflects, in a finely resolved way, the difficulty of "solution checking." What happens if, more bluntly, we simply address fixed-parameter complexity for problems for which solutions can be checked in *polynomial* time? To study this question, as we mentioned earlier, it is natural to define the following complexity classes.

DEFINITION 3.9. *A parameterized problem L belongs to* $W[P]$ ($W[SAT]$) (*MONOTONE* $W[P]$) *if L uniformly reduces to the parameterized circuit problem* $L_{\mathcal{F}}$ *for some family of* (*Boolean*) (*monotone*) *circuits* $\mathcal{F}$.

Note that $W[t]$ is contained in $W[P]$ for every t, and that $W[P] = FPT$ if $P = NP$. With Karl Abrahamson, [1], [2], we have been able to show that all of the problems identified in Abrahamson et al. [3] as complete for PGT are uniformly complete for $W[P]$. (For the reasons mentioned before, we would argue that the present theory offers a better framework for those results and allows us to address much wider parameterized issues.) We have also identified a number of further natural complete problems. These results and some aspects of the structure of $W[P]$ as well as parameterized $PSPACE$ are reported in Abrahamson, Downey, and Fellows [1], [2]. For $PSPACE$ there are very interesting problems that seem hard and are natural ones that relate to winning strategies for k-move games.

4. Summary and open problems. We have presented in this paper a basic framework and fundamental completeness results for the study of fixed-parameter tractability. We view the exploration of this topic as a large project, of which this constitutes only the initial step. As can be seen from the appendix and the reference list, in the time that this paper has been in the refereeing/publication process, there has already been quite a bit of work using our classification. We believe that our techniques are of particular interest in the area of molecular biology, and because of the fact that although problems may have no polynomial time approximation scheme unless $P = NP$, they can certainly still be in FPT.

In some ways, the study of fixed-parameter tractability and completeness addresses the subject of computational infeasibility inside of P. For related work from a different perspective see Buss and Goldsmith [35] and the references cited there. Many of the approaches and issues concerning the standard complexity classes have natural analogues in this setting that are thus far unexplored.

Consider, for example, the issue of parallel complexity. Trivially, there is a parallel algorithm running in time $O(\log n)$ and using n^k processors to determine if a graph G on n vertices has a dominating set of size k, for each fixed k. For a contrasting result, Lagergren [95] has shown that for each fixed k, it can be determined in time $O(\log^3 n)$ with $O(n)$ processors whether a graph has treewidth at most k. This suggests a natural fixed-parameter analogue of NC. Similar remarks apply to randomized complexity. With Ken Regan, the authors have made a little progress with randomized complexity [64].

For another example, consider approximation algorithms. One of the fundamental results of Robertson and Seymour (apart from their work on graph minors) is that there is an algorithm that in time $f(k) \cdot n^2$ finds, for a graph G of order n, either (1) a tree decomposition of width at most $5k$, or (2) evidence that the treeewidth of G is greater than k. (Of course this is now replaced by Bodlaender's linear time algorithm [23].) An analogous result for DOMINATING SET might be an algorithm running in time $f(k) \cdot n^c$ that finds either (1) a dominating set

of size $O(k)$, or (2) evidence that the minimum size of a dominating set for G is greater than k. Such an algorithm is presently unknown. It may even be that the existence of such an algorithm would imply the collapse of the W-Hierarchy, much as the existence of a P-time relative approximation algorithm for the TRAVELING SALESPERSON problem would imply $P = NP$ (Garey and Johnson [81]).

Many interesting structural questions concerning the W-Hierarchy remain to be explored. For instance we have established some connections between classical classes and our parameterized classes. Thus if $W[2t] \subseteq FPT$, then $2t$-NORMALIZED SATISFIABILITY is soluable in time $p(n)2^{o(v)}$, where n is the size of the input, p is a polynomial, and v is the number of variables. For this result and other structural results along these lines, see Abrahamson, Downey, and Fellows [2]. Nevertheless, the precise relationship between classical classes and the parameterized one is still unclear. We also are not aware of an oracle separating the W-Hierarchy, nor do we know if collapse propogates upwards. (That is, if $FPT = W[t]$ implies $FPT = W[t+1]$, for instance.)

From a concrete point of view, we also do not know if a problem such as the following belongs to $W[t]$ for any t.

TWO PLAYER DOMINATING SET
Instance: A graph $G = (V, E)$ and a positive integer k.
Question: Is it true that for every k-element subset $V' \subseteq V$, there is a k-element subset $V'' \subseteq V$ such that $V' \cup V''$ is a $2k$-element dominating set for G?

We have been able to prove the following density theorem.

THEOREM 4.1. *For the strong uniform reduction hierarchy, if any of the containments*

$$FPT \subseteq W[1] \subseteq W[2] \subseteq \cdots$$

is proper, then there are infinitely many intervening equivalence classes of parameterized problems with respect to strong uniform reductions.

Actually, we can prove a much stronger result along the lines of the full Ladner [94] theorem. Also Downey and Fellows [58] contains quite a number of other structural and relativization results. For instance we know that there is an oracle with $P \neq NP$ yet the W-Hierarchy collapses. It is an open question whether an analogue of Theorem 4.1 holds in the uniform case. We remark that the setting of parameterized problems introduces some technical challenges for density results. Our proof of Theorem 4.1 [58] employs techniques from the infinite-injury priority method. Technically while the standard polynomial time reductions are Σ_2 (on recursive languages), the fact is that strong uniform reducibility is Σ_3-complete, and the other two reductions are Σ_4-complete [61]. (The last result needs a tree of strategies infinite injury priority argument.)

Finally, we think the primary value of our theory of fixed-parameter tractability is that there is, for many parameterized problems, a compelling practical interest. There are many natural parameterized problems that may well be complete for various levels of the W-Hierarchy. Demonstrations of such completeness would provide an explanation of why, although they are soluable in polynomial time for each fixed parameter value, these problems resist attempts to show fixed-parameter tractability.

5. Appendix: A problem compendium and guide to W-Hierarchy completeness, hardness, and classification, and some open questions. This appendix contains problem definitions and summaries of most of the presently known completeness and hardness results, and information concerning fixed-parameter tractability for restrictions of problem instances. References are given where appropriate. The problems discussed are grouped (more or less)

according to level in the W-Hierarchy. Our list for FPT is obviously incomplete, but the given examples should provide the reader with the flavour of such results.

5.1. In FPT.

Remark. The following is only a small list of the known FPT problems. Many more can be obtained by Courcelle's theorem applied to classes of bounded tree width, in conjunction with Bodlaender's theorem on Treewidth k recognition. (See e.g., Abrahamson and Fellows [4], [5], Courcelle [49], Downey and Fellows [61], or Van Leeuwen [115].)

CROSSING NUMBER FOR MAX DEGREE 3 GRAPHS
Instance: A graph G all of whose vertices have max degree 3.
Parameter: A positive integer k.
Question: Does G have a an embedding with crossing number $\leq k$?

This is $O(n^3)$ by [69] via [106], [107].

ALTERNATING HITTING SET
Instance: A collection C of subsets of a set B with $|c| \leq k_1$ for all $c \in C$, an integer k_2.
Parameter: A positive integer kk_1, k_2.
Question: Does player one have a forced win in $\leq k_2$ moves in the following game played on C and B? Players alternate choosing a new element of B until, for each $c \in C$, some member of c has been chosen. The player whose choice causes this to happen loses.

The general version of this problem is *PSPACE*-complete by a reduction from *QBF* (see Garey and Johnson [81, Bp7]). This problem is in FPT by Abrahamson, Downey, and Fellows [2]. Soluable in $O(n)$ time for fixed k_1 and k_2.

CUTWIDTH
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Is the cutwidth of $G \leq k$?

The general version of this problem is *NP*-complete by a reduction from SIMPLE MAX CUT (see Garey and Johnson [81, GT44]). This problem is in FPT by Fellows and Langston [75]. Soluable in $O(n)$ time for fixed k (Bodlaender [17]).

DIAMETER IMPROVEMENT FOR PLANAR GRAPHS
Instance: A planar graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Can G be augmented with additional edges in such a way that the resulting graph G' remains planar and the diameter of G' is at most k?

This problem is in FPT by Downey and Fellows [59] after Robertson and Seymour [106]. Soluable in $O(n)$ time for fixed k (Bodlaender [17]).

MINIMUM FILL-IN
Input: A graph G.
Parameter: A positive integer k.
Question: Can we add $\leq k$ edges to G and cause G to become chordal?

The general problem is *NP* complete by Yannakakis [119]. Soluable in time $O(c^k.|E|)$ and $O(k^5|E||V| + f(k))$ by Kaplan, Shamir, and Tarjan [91].

DISJOINT PATHS
Instance: A graph $G = (V, E)$, $s_1, \ldots, s_k \in V$ *start* vertices, $t_1, \ldots, t_k \in V$ *end* vertices.

Parameter: k
Question: Do there exist vertex disjoint paths $P_1, \ldots, P_k$ such that P_i starts at vertex s_i and ends at vertex t_i for $i = 1 \ldots k$?

The general version of this problem is *NP*-complete by a reduction from 3SAT (see Garey and Johnson [81, ND40]). This problem is in FPT by Robertson and Seymour [106]. Soluable in $O(n^3)$ time for fixed k.

FEEDBACK VERTEX SET
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Is there a set U of k vertices of G such that each cycle of G passes through some vertex of U?

The general version of this problem is *NP*-complete by a reduction from VERTEX COVER (see Garey and Johnson [81, GT7]). This problem is in FPT by Downey and Fellows [59] and Bodlaender [16]. Soluable in $O(n)$ time for fixed k.

GATE MATRIX LAYOUT
Instance: A boolean matrix M.
Parameter: A positive integer k.
Question: Is there a permutation of the columns of M so that, if in each row we change to $*$ every 0 lying between the row's leftmost and rightmost 1, then no column contains more than k 1's and $*$'s?

This problem is in FPT by Fellows and Langston [77]. Soluable in $O(n)$ time for fixed k Bodlaender [17]. Equivalent to GRAPH PATHWIDTH.

GRAPH GENUS
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Does G have genus k?

The general version of this problem is *NP*-complete. This problem is in *FPT* by Fellows and Langston [74] via Robertson and Seymour [106]. Soluable in time $O(n^3)$ for fixed k by the results of Robertson and Seymour.

LONG CYCLE
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Does G have a cycle of length $\geq k$?

The general version of this problem is *NP*-complete by a reduction from HAMILTON CIRCUIT (see Garey and Johnson [81, ND28]). This problem is in FPT by Fellows and Langston [74]. Soluable in $O(n)$ time for fixed k.

MAX LEAF SPANNING TREE
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Does G have a spanning tree with k or more leaves?

The general version of this problem is *NP*-complete by a reduction from DOMINATING SET (see Garey and Johnson [81, ND2]). This problem is in FPT by Downey, and Fellows [59] and Bodlaender [16]. In $LOGSPACE + Advice$ by Cai et al. [41]. Soluable in $O(n)$ for fixed k.

MINIMUM DISJUNCTIVE NORMAL FORM
Instance: A set $X = \{x_1, x_2, \ldots, x_n\}$ of variables, a set $A \subseteq \{0, 1\}^n$ of implicants, a positive interger m.
Parameter: $|A|$.
Question: Is there a DNF expression E over X, having no more than m disjuncts, such that E is true for precisely those truth assignments in A and no others?

The general version of this problem is *NP*-complete by a reduction from SET COVER (see Garey and Johnson [81, LO9]). This problem is in FPT by [66]. Soluable in $O(2^{|A|}n)$ time for fixed A.

MINOR ORDER TEST
Instance: Graphs $G = (V, E)$ and $H = (V', E')$.
Parameter: H
Question: Is $H \leq_{minor} G$?

The general version of this problem is *NP*-complete by a reduction from HAMILTON CIRCUIT (see Garey and Johnson [81, OPEN2]) This problem is in FPT by Robertson and Seymour [106]. Soluable in $O(n^3)$ time for fixed k.

PLANAR FACE COVER
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Can G be embedded in the plane so that there are k faces which cover all vertices?

The general version of this problem is *NP*-complete by a reduction from VERTEX COVER (Fellows [70]). This problem is in *FPT* by Bienstock and Monma [15]. Soluable in $O(n)$ time for fixed k.

SEARCH NUMBER
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Are k searchers sufficient to ensure the capture of a fugitive who is free to move with arbitrary speed about the edges of G?

The general version of this problem is *NP*-complete. This problem is in FPT by Fellows and Langston [75]. Soluable in $O(n)$ time for fixed k (Bodlaender [17]).

STEINER TREE
Instance: A graph $G = (V, E)$, a set S of at most k vertices in V, an integer m.
Parameter: k
Question: Is there a set of vertices $T \subseteq V - S$ such that $|T| \leq m$ and $G[S \cup T]$ is connected?

The general version of this problem is *NP*-complete by a reduction from EXACT COVER (see Garey and Johnson [81, ND12]) [82]. This problem is in FPT by Dreyfus and Wagner [65]. Soluable in time $O(3^k n + 2^k n^2 + n^3)$ by the Dreyfus–Wagner algorithm (Wareham [117]).

TREEWIDTH
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Does G have treewidth k?

The general version of this problem is *NP*-complete [8]. Soluable in $O(n)$ time for fixed k (Bodlaender [23]).

VERTEX COVER
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Does G have a vertex cover of size $\leq k$?

The general version of this problem is *NP*-complete by a reduction from 3SAT (see Garey and Johnson [81, GT1]). This problem is in FPT by Downey and Fellows [59] and Buss [34]; and in fact in $LOGSPACE + Advice$ by Cai et al. [41]. Soluable in $O(2^k n)$ or $O(n + k^k)$ time.

k-PERFECT MATCHINGS
Input: A graph G.
Parameter: A positive integer k.
Question: Does G have at least (or exactly) k perfect matchings?

The general problem of finding the maximum number of perfect matchings is $\#P$ complete (in the size of G) even for bipartite graphs by Valiant [113]. For $k = 1$, the problem is in P for bipartite graphs by the old work of Ford and Fulkerson, and for general graphs by the work of Edmonds. The problem is $O(k.e)$, where e denotes the number of edges, for any fixed k by Itai, Rodeh, and Tanimoto [89]. For $weighted$ graphs one finding the best k matchings is FPT by, for instance, Chegireddy and Hamacher [46].

SHORT 3DIMENSIONAL MATCHING
Input: A graph $G \subseteq X \times Y \times Z$ with $|X| = |Y| = |Z|$.
Parameter: A positive integer k.
Question: Does there exist a subset $G' \subseteq G$ such that $|G'| = k$ and for all $\langle x, y, z\rangle \neq \langle x', y', z'$ both in G', we have $x \neq x'$, $y \neq y'$, and $z \neq z'$?

The general problem with k varying is *NP* complete and is one of Garey and Johnson's six basic problems. For k fixed the problem is FPT by Downey and Fellows [61].

5.2. In FPT(nonuniform).

GRAPH LINKING NUMBER
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Can G be embedded into 3-space such that the maximum size of a collection of topologically linked disjoint cycles is bounded by k?

This problem is in FPT(nonuniform) by Fellows and Langston [74] after Robertson and Seymour [106]. Soluable in $O(n^3)$ time for fixed k.

5.3. In randomized FPT.

BOUNDED FACTOR FACTORIZATION
Instance: An n-bit positive integer N.
Parameter: A positive integer k.
Question: Is there a prime factor p of N such that $p < n^k$?

This problem is in randomized FPT by Fellows and Koblitz [72], [73].

LINEAR EXTENSION COUNT
Instance: A poset $(P, \leq)$.
Parameter: A positive integer k.
Question: Does P have at least k linear extensions?

This problem is in randomized FPT by Brightwell and Winkler [32], [33]. Not known to be in FPT.

POLYNOMIALLY SMOOTH NUMBER
Instance: An n-bit positive integer N.
Parameter: A positive integer k.
Question: Is N n^k–smooth, i.e., is every prime divisor of N bounded by n^k?

This problem is in randomized FPT by Fellows and Koblitz [72], [73]. n^k–smoothness of n-digit numbers is a natural number-theoretic property that arises in the study of polynomial-time complexity. For example, the concept plays a central role in the demonstration that primality is in $UP \cap co\text{-}UP$.

SMALL PRIME DIVISOR
Instance: An n-bit positive integer N.
Parameter: A positive integer k.
Question: Does N have a nontrivial divisor less than n^k?

This problem is in randomized FPT by Fellows and Koblitz [72], [73].

5.4. *W*[1]-complete.

t THRESHOLD STABLE SET
Instance: A directed graph $G = (V, A)$.
Parameter: A positive integer k.
Question: Does G have a stable set of size k? (A *stable set* is a set of vertices $V' \subseteq V$ such that for every vertex v of $V - V'$, there are fewer than t vertices $u \in V'$ with $uv \in A$.)

This problem is $W[1]$-complete by a reduction from INDEPENDENT SET. Complete for $W[1]$ by Downey and Fellows [61].

BINARY CLADISTIC CHARACTER COMPATIBILITY
Instance: A set C of n binary cladistic characters over m objects.
Parameter: A positive integer k.
Question: Is there a subset $C' \subseteq C$, $|C'| = k$, such that all pairs of characters in C' are compatible?

The general version of this problem is *NP*-complete by a reduction from CLIQUE (Day and Sankoff [52]). This problem is $W[1]$-complete by the same reduction (Wareham [117]). The unconstrained-character version of this problem is also $W[1]$-complete (Wareham [117]). If $k = |C|$, one obtains the problem TRIANGULATING COLORED GRAPHS (PERFECT PHYLOGENY).

BINARY QUALITATIVE CHARACTER COMPATIBILITY
Instance: A set C of n binary qualitative characters over m objects.
Parameter: A positive integer k.
Question: Is there a subset $C' \subseteq C$, $|C'| = k$, such that all pairs of characters in C' are compatible?

The general version of this problem is *NP*-complete by a reduction from BINARY CLADISTIC CHARACTER COMPATIBILITY (Day and Sankoff [52], Wareham [117]). This problem is $W[1]$-complete by the same reduction. The unconstrained-character version of this problem is $W[1]$-hard (Wareham [117]).

CLIQUE
Instance: A graph $G = (V, E)$, a positive integer k.
Parameter: A positive integer k.
Question: Is there a set of k vertices $V' \subseteq V$ that forms a complete subgraph of G (that is, a clique of size k)?

The general version of this problem is *NP*-complete by a reduction from VERTEX COVER (see Garey and Johnson [81, GT19]). This problem is $W[1]$-complete by a reduction from INDEPENDENT SET by Downey and Fellows [56]. Fixed-parameter tractable for planar graphs and for graphs of maximum degree $f(k)$ for any fixed function f.

INDEPENDENT SET
Instance: A graph $G = (V, E)$, a positive integer k.
Parameter: A positive integer k.
Question: Is there a set $V' \subseteq V$ of cardinality k, such that $\forall u, v \in V', uv \notin E$?

The general version of this problem is *NP*-complete by a reduction from VERTEX COVER (see Garey and Johnson [81, GT20]). This problem is $W[1]$-complete by a reduction from ANTIMONOTONE $W[1, 2]$ by Downey and Fellows [56]. Fixed-parameter tractable for planar graphs.

LONGEST COMMON SUBSEQUENCE (I)
Instance: A set of k strings $X_1, \ldots, X_k$ over an alphabet Σ, a positive integer m.
Parameter: k, m
Question: Is there a string $X \in \Sigma^*$ of length at least m that is a subsequence of X_i for $i = 1, \ldots, k$?

The general version of this problem is *NP*-complete by a reduction from VERTEX COVER (see Garey and Johnson [81, SR10]). This problem is $W[1]$-complete by a reduction from CLIQUE by Bodlaender et al. [25].

d-RED/BLUE NONBLOCKER
Instance: A graph $G = (V, E)$ of maximum degree d where V is partitioned into two color classes $V = V_{\text{red}} \cup V_{\text{blue}}$.
Parameter: A positive integer k.
Question: Is there is a set of red vertices $V' \subseteq V_{\text{red}}$ of cardinality k such that every blue vertex has at least one neighbor that does not belong to V'?

This problem is $W[1]$-complete by a reduction from $W[1, s]$ (Downey and Fellows [56]).

SEMIGROUP EMBEDDING
Instance: A semigroup $(S, \cdot)$.
Parameter: A positive integer k and a semigroup (H, x).
Question: Can H be embedded into S?

This problem is $W[1]$-complete by a reduction from CLIQUE (Downey and Fellows [61]).

SEMILATTICE EMBEDDING
Instance: A semilattice L.
Parameter: A positive integer k and a semilattice H.
Question: Is H embeddable into L?

This problem is $W[1]$-complete by a reduction from CLIQUE (Downey and Fellows [61]).

SET PACKING
Instance: A finite family of sets $S = S_1, \ldots, S_n$, an integer k.
Parameter: k
Question: Does S contain a subset R of k mutually disjoint sets?

The general version of this problem is *NP*-complete by a reduction from X3C (see Garey and Johnson [81, SP3]). This problem is $W[1]$-complete by a reduction from INDEPENDENT SET (Ausiello, D'Atri, and Protasi [11], Wareham [117]).

SHORT CONTEXT SENSITIVE DERIVATION
Instance: A context-sensitive grammar $G = (N, \Sigma, \Pi, S)$, a word $x \in \Sigma^*$.
Parameter: A positive integer k.
Question: Is there a G-derivation of x of length at most k?

The general version of this problem is $PSPACE$-complete by a reduction from LINEAR BOUNDED AUTOMATON ACCEPTANCE (see Garey and Johnson [81, AL20]). This problem is $W[1]$-complete by a reduction from CLIQUE (Downey et al. [62], [63].)

SHORT TURING MACHINE ACCEPTANCE
Instance: A nondeterministic Turing machine M operating on alphabet Σ, a word $x \in \Sigma^*$.
Parameter: A positive integer k.
Question: Is there a computation of M on input x that reaches an accept state in at most k steps?

The general version of this problem is undecidable (see for example Hopcroft and Ullman [88]). This problem is $W[1]$-complete by a reduction from CLIQUE (Downey et al. [63], Cesati [44]). In FPT if either the size of the alphabet or the number of nondeterministic transition possibilities out of a given state is bounded.

SHORT POST CORRESPONDENCE
Instance: A Post system Π.
Parameter: A positive integer k.
Question: Is there a length k solution for Π?

The classical POST CORRESPONDENCE problem is a well-known undecidable problem. This problem is $W[1]$-complete by a reduction from SHORT UNRESTRICTED GRAMMAR DERIVATION (Cai et al. [40].)

SHORT UNRESTRICTED GRAMMAR DERIVATION
Instance: An unrestricted phrase-structure grammar G, a word x.
Parameter: A positive integer k.
Question: Is there a G-derivation of x of length at most k?

The general version of this problem is undecidable (see for example Hopcroft and Ullman [88]. This problem is $W[1]$-complete by a reduction from CLIQUE (Cai et al. [40]).

SQUARE TILING
Instance: A set C of "colors," a collection $T \subseteq C^4$ of "tiles" (where $\langle a, b, c, d\rangle$ denotes a tile whose top, right, bottom, and left sides are colored a, b, c, and d, respectively), a positive integer $k \leq C$.
Parameter: k
Question: Is there a tiling of an $k \times k$ square using the tiles in T, i.e., an assignment of a tile $A(i, j) \in T$ to each ordered pair i, j, $1 \leq i \leq k$, $1 \leq j \leq k$, such that (1) if $f(i, j) = \langle a, b, c, d\rangle$ and $f(i+1, j) = \langle a', b', c', d'\rangle$, then $a = c'$, and (2) if $f(i, j) = \langle a, b, c, d\rangle$ and $f(i, j+1) = \langle a', b', c', d'\rangle$, then $b = d'$.

The general version of this problem is *NP*-complete by a reduction from DIRECTED HAMILTON PATH (see Garey and Johnson [81, GP13]). This problem is $W[1]$-complete by Cai et al. [40], Downey and Fellows [61].

VAPNIK-CHERVONENKIS (VC) DIMENSION
Instance: A family of subsets F of a base set X.
Parameter: A positive integer k.
Question: Is the VC dimension of F at least k? (The VC dimension of a family of subsets F of a base set X is the maximum cardinality of a set $S \subseteq X$ such that for each subset $S' \subseteq S$, $\exists Y \in F$ such that $S \cap Y = S'$.)

The general version of this problem is $LOGSNP$-complete (Papadimitriou and Yannakakis [100]). This problem is $W[1]$-complete by a reduction from CLIQUE (Downey, Evans, and Fellows [54]). Membership of $W[1]$ is proven by a generic reduction in [59].

WEIGHTED q-CNF SATISFIABILITY
Instance: A q-CNF formula X, i.e., a CNF formula such that each clause has no more than q literals.
Parameter: A positive integer k.
Question: Does X have a satisfying assignment of weight k?

This problem is $W[1]$-complete by a reduction from INDEPENDENT SET (Downey and Fellows [56]).

5.5. W[1]-hard, in W[2].

PERFECT CODE
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Does G have a k-element perfect code? (A *perfect code* is a set of vertices $V' \subseteq V$ with the property that for each vertex $v \in V$ there is precisely one vertex in $N[v] \cap V'$.)

This problem is $W[1]$-hard by a reduction from INDEPENDENT SET and in $W[2]$ by (Downey and Fellows [56]). We believe that it may be difficult intermediate between $W[1]$ and $W[2]$.

WEIGHTED EXACT CNF SATISFIABILITY
Instance: A boolean expression E in conjunctive normal form.
Parameter: A positive integer k.
Question: Is there a truth assignment of weight k to the variables of E that makes exactly one literal in each clause of E true?

This problem is $W[1]$-hard by a reduction from PERFECT CODE and in $W[2]$ by (Downey and Fellows [56]). Equivalent to PERFECT CODE (Downey and Fellows [56]). A related problem is UNIQUE WEIGHTED CNF SATISFIABILITY below.

UNIQUE WEIGHTED CNF SATISFIABILITY
Instance: A boolean CNF formula X.
Parameter: A positive integer k.
Question: Is there a unique weight k satisfying assignment for X?

Clearly this is in $W[2]$. There are obvious versions of the above for normalized satisfiability at any level of the W-Hierarchy. For the CNF situation above, the problem is clearly in the natural analogue of D_p. The reader should recall that D_p consists of the class of languages L that can be expressed as the interchapter of a language in *NP* and one in *co-NP*. Clearly we

can similarly define $D_p[2]$ (or, more generally $D_p[t]$) as for D_p but with $W[2]$ in place of NP. Then this problem is in $D_p[2]$, as we see below. Before we prove this we remark that using the Valiant–Vazirani technique [114] we can show that UNIQUE WEIGHTED CNF SATISFIABILITY for arbitary boolean formulae is the same as WEIGHTED CNF SATISFIABILITY for boolean formulae under randomized reductions. However, on the face of it, it is not clear if this is true for any finite level of the W-Hierarchy since for instance weight is lost in the [VV] proof. Nevertheless using a new argument based on coding theory (Hadamard codes) together with Ken Regan the authors [64] have shown that a weighted version of [114] holds for all $t \geq 2$. This seems a very fruitful area to analyse.

UNIQUE WEIGHTED CNF SATISFIABILITY is in $D_p[2]$

It suffices to describe how to say that a CNF expression has at least two satisfying expressions in $W[2]$. Let C be the circuit corresponding to X. Take two copies of C. Add $O|X|$ many gates to express the fact that the first copy of C has a satisfying assignment different from the second. Now for k choose q and r appropriately and take q copies of the the left circuit and r copies of the right. Add new gates to express the fact that the inputs of the left q must all be equal and the fact that the inputs of the right r must all be equal. Now accept C if the new circuit has a weight $(q + r)k$ accepting input. Then for the correct choice of (q, r), depending only on k, C has two or more accepting inputs if and only if the new circuit has one of weight $(q + r)k$.

We remark that this seems to be where UNIQUE DOMINATING SET would lie. We do not at present know if there are $D_p[t]$ complete problems.

5.6. W[1]-hard, in W[P].

PERMUTATION GROUP FACTORIZATION
Instance: A set A of permutations $A \subseteq S_n$, $x \in S_n$.
Parameter: A positive integer k.
Question: Does x have a factorization of length k over A?

This problem is $W[1]$-hard by a reduction from PERFECT CODE and in $W[P]$ by Cai et al. [40].

SUBSET SUM
Instance: A set of integers $X = \{x_1, \ldots, x_n\}$, integers s, k.
Parameter: k
Question: Is there a subset $X' \subseteq X$ of cardinality k such that the sum of the integers in X' equals s?

The general version of this problem is NP-complete by a reduction from *Partition* (see Garey and Johnson [81, SP13]). This problem is $W[1]$-hard by a reduction from PERFECT CODE Downey and Fellows [56] and in $W[P]$ by Fellows and Koblitz [73]. Not known to belong to $W[t]$ for any t.

5.7. W[1]-hard.

α-BALANCED SEPARATOR
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Does there exist a set of vertices S, $|S| \leq k$, such that every component of $G[V - S]$ has at most $\alpha|V|$ vertices?

This problem is $W[1]$-hard by a reduction from CLIQUE by Kaplan and Shamir [90]. This problem is $W[1]$-hard for every fixed α.

COLORED PROPER INTERVAL GRAPH COMPLETION
Instance: A graph $G = (V, E)$, a vertex coloring $c : V \to \{1, \ldots, k\}$.
Parameter: k
Question: Does there exist a proper interval supergraph of G which respects c?

This problem is $W[1]$-hard by a reduction from INDEPENDENT SET by Kaplan and Shamir [90].

COLORED UNIT INTERVAL GRAPH COMPLETION
Instance: A graph $G = (V, E)$, a vertex coloring $c : V \to \{1, \ldots, k\}$.
Parameter: k
Question: Does there exist a graph $G' = (V, E')$ such that $E' \supseteq E$, G' is a unit interval graph and G' is properly colored by c?

The general version of this problem is *NP*-complete by Kaplan and Shamir [90]. This problem is $W[1]$-hard by Kaplan and Shamir [90].

EXACT CHEAP TOUR
Instance: A weighted graph $G = (V, E)$, a weight function $w : E \to \mathbf{Z}$.
Parameter: A positive integer k.
Question: Is there a tour through at least k nodes of G of cost exactly S?

The general version of this problem is *NP*-complete by a reduction from HAMILTON CIRCUIT (see Garey and Johnson [81, ND22]). This problem is $W[1]$-hard by [56]. See the related SHORT CHEAP TOUR problem.

INVMAX
Instance: A circuit C, an initial configuration conf_0 describing the placing of inverters on connections between gates in G.
Parameter: A positive integer k.
Question: Is there a subset A of the gates in G to which DeMorgan's rules can be applied such that the resulting circuit will have at least k gates without any inverters attached to their output lines?

The general version of this problem is *NP*-complete by a reduction from INDEPENDENT SET (Simon [109]). This problem is $W[1]$-hard by the same reduction.

PROPER INTERVAL SANDWICH WITH BOUNDED CLIQUE SIZE
Instance: A sandwich instance $S = (V, E^1, E^3)$.
Parameter: A positive integer k.
Question: Does there exist a sandwich G for S which is a proper interval graph such that the size of largest clique is at most k?

The general version of this problem is *NP*-complete by Kaplan and Shamir [90]. This problem is $W[1]$-hard by a reduction from COLORED INTERVAL GRAPH COMPLETION by Kaplan and Shamir [90].

REACHABILITY DISTANCE FOR VECTOR ADDITION SYSTEMS (PETRI NETS)
Instance: A set T of m length n integer-valued vectors $T = \{x^i = (x_1^i, \ldots, x_n^i) : 1, \leq i \leq m\}$, a nonnegative starting vector $s = (s_1, \ldots, s_n)$, a nonnegative target vector $t = (t_1, \ldots, t_n)$.
Parameter: A positive integer k.
Question: Is there a choice of k indices $i_1, \ldots, i_k$, $1 \leq i_j \leq m$ for $j = 1, \ldots, k$ such that $t = s + \sum_{j=1}^{k} x^{i_j}$ and such that every intermediate sum is nonnegative in each component, that is, $s_r + \sum_{j=1}^{q} x_r^{i_j} \geq 0$ for $q = 1, \ldots, k$ and $r = 1, \ldots, n$?

This problem is $W[1]$-hard by a reduction from CLIQUE (Downey et al. [63]).

SHORT l TAPE NDTM COMPUTATION (I)
Instance: An l-tape nondeterministic Turing machine M operating on alphabet Σ, a word $x \in \Sigma^*$.
Parameter: A positive integer k.
Question: Is there a computation of M on input x that reaches an accept state in at most k steps?

The general version of this problem is undecidable (see Hopcroft and Ullman [88]). This problem is $W[1]$-hard by a reduction from SHORT TURING MACHINE COMPUTATION (Cesati [44]).

SHORT l TAPE NDTM COMPUTATION (II)
Instance: An l-tape nondeterministic Turing machine M operating on alphabet Σ, a word $x \in \Sigma^*$, a positive integer k.
Parameter: k, l
Question: Is there a computation of M on input x that reaches an accept state in at most k steps?

The general version of this problem is undecidable (see Hopcroft and Ullman [88]). This problem is $W[1]$-hard by a reduction from SHORT TURING MACHINE COMPUTATION (Cesati[44]).

SUBSET PRODUCT
Instance: A set of integers $X = \{x_1, \ldots, x_n\}$, integers a, m, k.
Parameter: k
Question: Is there a subset $X' \subseteq X$ of cardinality k such that the product of the integers in X' is congruent to a mod m?

The general version of this problem is *NP*-complete by a reduction from X3C (see Garey and Johnson [81, SP14]). This problem is $W[1]$-hard by a reduction from PERFECT CODE by Fellows and Koblitz [72], [73].

COLORED GRAPH AUTOMORPHISM
Instance: A 2-colored (bipartite) graph G.
Parameter: A positive integer k.
Question: Is there an automorphism preserving colors moving exactly k blue vertices?

$W[1]$-hard by a reduction from ANTIMONOTONE 2SAT (Downey and Fellows [61]).

5.8. *W*[2]-complete.

DOMINATING SET
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Is there a set of k vertices $V' \subseteq V$ with the property that every vertex of G either belongs to V' or has a neighbor in V'?

The general version of this problem is *NP*-complete by a reduction from VERTEX COVER (see Garey and Johnson [81, GT2]). This problem is $W[2]$-complete by a reduction from WEIGHTED CNF SATISFIABILITY (Downey and Fellows [55], this paper). Fixed-parameter tractable for planar graphs (Downey and Fellows [59]). Problem is $W[2]$-hard if the dominating set V' is required to be either connected or *total*, i.e., for each vertex in V there is an edge to some vertex in V' (Bodlaender and Kratsch [30].

HITTING SET
Instance: A finite family of sets $S = S_1, \ldots, S_n$ comprised of elements from $U = \{u_1, \ldots, u_m\}$.
Parameter: A positive integer k.
Question: Is there a subset $T \subseteq U$ of size k such that for all $S_i \in S$, $S_i \cap T \neq \emptyset$?

The general version of this problem is *NP*-complete by a reduction from VERTEX COVER (see Garey and Johnson [81, SP8]). This problem is $W[2]$-complete by a reduction from SET COVER (Ausiello, D'Atri, and Protasi [11], Wareham [117]).

INDEPENDENT DOMINATING SET
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Is there a set of k vertices $V' \subseteq V$ that is both an independent set and a dominating set in G?

The general version of this problem is *NP*-complete by a reduction from VERTEX COVER (see Garey and Johnson [81, GT2]). This problem is $W[2]$-complete by Downey and Fellows [55]. Fixed-parameter tractable for planar graphs (Downey and Fellows [59]).

SET COVER
Instance: A finite family of sets $S = S_1, \ldots, S_n$.
Parameter: A positive integer k.
Question: Is there a subset $R \subseteq S$ whose union is all elements in the union of S?

The general version of this problem is *NP*-complete by a reduction from X3C (see Garey and Johnson [81, SP5]). This problem is $W[2]$-complete by a reduction from DOMINATING SET by Paz and Moran [101], Wareham [117].

TOURNAMENT DOMINATING SET
Instance: A tournament T.
Parameter: A positive integer k.
Question: Does T have a dominating set of cardinality at most k?

The general version of this problem is $LOGSNP$-complete (Papadimitriou and Yannakakis [100]). This problem is $W[2]$-complete by a reduction from DOMINATING SET (Downey and Fellows [59]).

WEIGHTED BINARY INTEGER PROGRAMMING
Instance: A binary matrix A, a binary vector $\mathbf{b}$.
Parameter: A positive integer k.
Question: Does $A \cdot \mathbf{x} \geq \mathbf{b}$ have a binary solution of weight k?

The general version of this problem is *NP*-complete by a reduction from 3SAT (see Garey and Johnson [81, MP1]). This problem is $W[2]$-complete by a reduction from MONOTONE WEIGHTED CNF SATISFIABILITY (Downey and Fellows, this paper). The problem WEIGHTED EXACT BINARY INTEGER PROGRAMMING asks that equality hold is hard for $W[1]$.

5.9. *W*[2]-hard, in *W*[*P*].

MONOCHROME CYCLE COVER
Instance: An edge-colored graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Is there a set of k vertices $V' \subseteq V$ with the property that every monochrome cycle in G contains a vertex in V'?

This problem is $W[2]$-hard and in $W[P]$ by Downey and Fellows [61]. Not known to belong to $W[t]$ for any t, but easily shown to be in $W[P]$.

MONOID FACTORIZATION
Instance: A set A of self-maps on $[n]$, a self-map h.
Parameter: A positive integer k.
Question: Is there a factorization of h of length k over A?

This problem is $W[2]$-hard by a reduction from DOMINATING SET and in $W[P]$ by Cai et al. [40].

5.10. *W*[2]-hard.

DOMINATING CLIQUE
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Is there a set of k vertices $V' \subseteq V$ that forms a complete subgraph of G and is also a dominating set for G?

This problem is $W[2]$-hard by a reduction from DOMINATING SET (Bodlaender and Kratsch [30]). Problem is in FPT if V' is also required to be *efficient*, i.e., each vertex not in V' is dominated by exactly one vertex in V' (Bodlaender and Kratsch [30]).

LONGEST COMMON SUBSEQUENCE II
Instance: A set of k strings $X_1, \ldots, X_k$ over an alphabet Σ, a positive integer m.
Parameter: k, m
Question: Is there a string $X \in \Sigma^*$ of length at least m that is a subsequence of X_i for $i = 1, \ldots, k$?

The general version of this problem is *NP*-complete by a reduction from VERTEX COVER (see Garey and Johnson [81, SR10]). This problem is $W[2]$-hard by a reduction from DOMINATING SET (Bodlaender, et al. [25]).

MAXIMAL IRREDUNDANT SET
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Is there a set $V' \subseteq V$ of cardinality k such that (1) each vertex $u \in V'$ has a private neighbor and (2) V' is not a proper subset of any $V'' \subseteq V$ which also has this property? (A *private neighbor* of a vertex $u \in V'$ is a vertex u' (possibly $u' = u$) with the property that for every vertex $v \in V'$, $u \neq v$, $u' \notin N[v]$.)

This problem is $W[2]$-hard by a reduction from DOMINATING SET (Bodlaender and Kratsch [30]). Originally proven to be $W[1]$ hard by Downey and Fellows (see [61]).

PRECEDENCE CONSTRAINED k-PROCESSOR SCHEDULING
Instance: A set T of unit-length tasks, a partial order $\prec$ on T, a positive integer deadline D, a number of processors k.
Parameter: k
Question: Is there a map $f : T \to \{1, \ldots, D\}$, such that for all $t, t' \in T$, $t \prec t'$ implies $f(t) < f(t')$, and for all i, $1 \leq i \leq D$, $|f^{-1}(i)| \leq k$?

The general version of this problem is Open (Garey and Johnson [81, OPEN8]). This problem is $W[2]$-hard by a reduction from DOMINATING SET (Bodlaender, Fellows, and Hallett [28]).

STEINER TREE
Instance: A graph $G = (V, E)$, a set S of at most k vertices in V, an integer m.
Parameter: k, m
Question: Is there a set of vertices $T \subseteq V - S$ such that $|T| \leq m$ and $G[S \cup T]$ is connected?

The general version of this problem is *NP*-complete by a reduction from EXACT COVER (see Garey and Johnson [81, ND12, GKR]). This problem is $W[2]$-hard by a reduction from DOMINATING SET (Bodlaender and Kratsch [30]).

5.11. $W[3]$-hard, in $W[4]$.

DOMINATING THRESHOLD SET
Instance: A graph $G = (V, E)$.
Parameter: Positive integers k, r.
Question: Is there a set $V' \subseteq V$ of at most k vertices such that for every vertex u, $N[u]$ contains at least r elements of V'?

This problem is $W[3]$-hard and in $W[4]$ by Fellows [69].

5.12. $W[t]$-complete.

WEIGHTED t-NORMALIZED SATISFIABILITY
Instance: A t-normalized boolean expression X.
Parameter: A positive integer k.
Question: Does X have a satisfying truth assignment of weight k?

This problem is $W[t]$-complete by Downey and Fellows (this paper).

$\leq k$ WEIGHTED t-NORMALIZED SATISFIABILITY
Instance: A boolean formula X.
Parameter: A positive integer k.
Question: Does X have a satisfying assignment of weight $\leq k$?

For $t \geq 2$ this problem is $W[t]$-complete by Cai and Chen [37]–[39]. This fact also follows by the main lemma of this paper. For $t = 1$ the problem is in FPT.

5.13. $W[t]$-hard, for all t, in $W[P]$.

SHORT PHONOLOGICAL SEGMENTAL DECODING
Instance: An integer k, a simplified segmental grammar $s = (F, S, D, R, c_p, C)$ such that the number of mutually exclusive rule sets in R, $|R_{m.e.}|$, is at most k, string $s \in S^+$.
Parameter: k
Question: Is there a string $u \in D$ such that $g(u) = s$?

The general version of this problem is *NP*-complete by a reduction from CLIQUE Ristad [105]. This problem is $W[t]$-hard by a reduction from WEIGHTED t-NORMALIZED SATISFIABILITY and in $W[P]$ by Downey et al. [63]. See Downey et al. [63] and Ristad [105] for definitions of simplified segmental grammars. The same proof also implies the $W[t]$-hardness and membership in $W[P]$ of SHORT PHONOLOGICAL SEGMENTAL ENCODING.

5.14. $W[t]$-hard, for all t.

BANDWIDTH
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.

Question: Is there a 1:1 linear layout $f : V \to \{1, \ldots, |V|\}$ such that $uv \in E$ implies $|f(u) - f(v)| \leq k$?

The general version of this problem is *NP*-complete by a reduction from 3PARTITION (see Garey and Johnson [81, ND40]). This problem is $W[t]$-hard by a reduction from UNIFORM EMULATION ON A PATH (Bodlaender, Fellows, and Hallett [28]). Remains $W[t]$-hard for all t when given graph is directed and layout must respect arc direction, or when given graph is a tree ([28]). The related problem CUTWIDTH is FPT (Fellows and Langston [74]).

COLORED CUTWIDTH
Instance: A graph $G = (V, E)$, an edge coloring $c : E \to \{1, \ldots, r\}$.
Parameter: A positive integer k.
Question: Is there a 1:1 linear layout $f : V \to \{1, \ldots, |V|\}$ such that for each color $j \in \{1, \ldots, k\}$ and for each i, $1 \leq i \leq |V| - 1$, we have $|\{uv : c(uv) = j \text{ and } f(u) \leq i \text{ and } f(v) \geq i + 1\}| \leq r$?

This problem is $W[t]$-hard by a reduction from LONGEST COMMON SUBSEQUENCE II (Bodlaender, Fellows, and Hallett [28]).

DOMINO TREEWIDTH
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Is the domino treewidth of G at most k?

The general version of this problem is *NP*-complete by a reduction from LONGEST COMMON SUBSEQUENCE II by Bodlaender and Engelfriet [27]. This problem is $W[t]$-hard by the same reduction.

FEASIBLE REGISTER ASSIGNMENT
Instance: A directed acyclic graph $G = (V, E)$, a positive integer k, a register assignment $r : V \to \{R_1, \ldots, R_k\}$.
Parameter: k
Question: Is there a linear ordering f of G, and a sequence $S_0, S_1, \ldots, S_{|V|}$ of subsets of V, such that $S_0 = \emptyset$, $S_{|V|}$ contains all vertices of in-degree 0 in G, and for all i, $1 \leq i \leq |V|$, $f^{-1}(i) \in S_i$, $S_i - \{f^{-1}(i)\} \subseteq S_{i-1}$ and S_{i-1} contains all vertices u for which $(f^{-1}(i), u) \in E$, and for all j, $1 \leq j \leq k$, there is at most one vertex $u \in S_i$ with $r(u) = R_j$?

The general version of this problem is *NP*-complete by a reduction from 3SAT (see Garey and Johnson [81, PO2]). This problem is $W[t]$-hard by a reduction from LONGEST COMMON SUBSEQUENCE II (Bodlaender, Fellows, and Hallett [28].)

INTERVALIZING COLORED GRAPHS (DNA PHYSICAL MAPPING)
Instance: A graph $G = (V, E)$, vertex coloring $c : V \to \{1, \ldots, k\}$.
Parameter: k
Question: Does there exist a supergraph $G' = (V, E')$ where $E \subseteq E'$ and G' is properly colored by c and is an interval graph?

The general version of this problem is *NP*-complete by a reduction from BETWEENNESS (Golumbic, Kaplan, and Shamir [85]) and INDEPENDENT SET (Fellows, Hallett, and Wareham [71]). This problem is $W[t]$-hard by a reduction from COLORED CUTWIDTH (Bodlaender, Fellows, and Hallett [28]). No polynomial time algorithm is known for fixed k.

LONGEST COMMON SUBSEQUENCE II
Instance: A set of k strings $X_1, \ldots, X_k$ over an alphabet Σ, a positive integer m.

Parameter: k
Question: Is there a string $X \in \Sigma^*$ of length at least m that is a subsequence of X_i for $i = 1, \ldots, k$?

The general version of this problem is *NP*-complete by a reduction from VERTEX COVER (see Garey and Johnson [81, SR10]). This problem is $W[t]$-hard by a reduction from MONOTONE WEIGHTED t-NORMALIZED SATISFIABILITY (Bodlaender, Fellows, and Hallett [28]).

LONGEST COMMON SUBSEQUENCE III
Instance: A set of k strings $X_1, \ldots, X_k$ over an alphabet Σ, a positive integer m.
Parameter: k, $|\Sigma|$
Question: Is there a string $X \in \Sigma^*$ of length at least m that is a subsequence of X_i for $i = 1, \ldots, k$?

The general version of this problem is *NP*-complete by a reduction from VERTEX COVER (see Garey and Johnson [81, SR10]). This problem is $W[t]$-hard by a reduction from Bodlaender et al. [24].

PROPER INTERVAL GRAPH COMPLETION PROBLEM WITH MINIMUM CLIQUE SIZE
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Does there exist a graph $G' = (V, E')$, $E' \supseteq E$, such that G' is an interval graph and has minimum clique size k?

The general version of this problem is *NP*-complete by Kaplan and Shamir [90]. This problem is $W[t]$-hard by Kaplan and Shamir [90]. Equivalent to the BANDWIDTH problem by Kaplan and Shamir [90] (also Hallett [87]).

PROPER INTERVAL GRAPH COMPLETION WITH BOUNDED CLIQUE SIZE
Input: A graph G.
Parameter: A positive integer k.
Question: Is there a $G' \supseteq G$ which is a proper interval graph and has clique size at most k?

The general problem is *NP*-complete and the parameterized problem hard for $W[t]$ for all t by Kaplan, Shamir, and Tarjan [91].

RESTRICTED COMPLETION TO A PROPER INTERVAL GRAPH WITH BOUNDED CLIQUE SIZE
Input: A graph G together with a set E of edges prohibited from G' below.
Parameter: A positive integer k.
Question: Is there a $G' \supset G$ which is a proper interval graph and has clique size at most k, and G' has no edges from E?

TRIANGULATING COLORED GRAPHS
Instance: A graph $G = (V, E)$, vertex coloring $c : V \to \{1, \ldots, k\}$.
Parameter: k
Question: Does there exist a supergraph $G' = (V, E')$ where $E \subseteq E'$ and G' is properly colored by c and G' is triangulated?

The general version of this problem is *NP*-complete by a reduction from INDEPENDENT SET (Bodlaender, Fellows, and Hallett [28]). This problem is $W[t]$-hard by a reduction from LONGEST COMMON SUBSEQUENCE II [28].

UNIFORM EMULATION ON A PATH
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Does there exist a function $f : V \to \{1, \ldots, |V|/k\}$ such that for all $uv \in E$ implies $|f(u) - f(v)| \leq 1$ and for all i, $|f^{-1}(i)| \leq k$?

This problem is $W[t]$-hard by a reduction from MONOTONE WEIGHTED t-NORMALIZED SATISFIABILITY (Bodlaender, Fellows, and Hallett [28]). Remains W[t]-hard for all t when given graph is a tree [28].

5.15. *W[P]*-complete.

k-BASED TILING
Instance: A tiling system with distinguished tiles.
Parameter: A positive integer k.
Question: Is there a tiling of the $n \times n$ plane using the tiling system and starting with exactly k distinguished tiles in a line ?

Reduction consists of a generic simulation of a Turing machine (see Downey and Fellows [61]).

k-INDUCED 3CNF SATISFIABILITY
Instance: A 3CNF formula φ.
Parameter: A positive integer k.
Question: Is there a set of k variables and a truth table assignment to those variables that causes φ to unravel?

This problem is $W[P]$-complete by a reduction from CHAIN REACTION CLOSURE (Abrahamson, Downey, and Fellows [2]) (also Abrahamson et al. [3]).

k-INDUCED SATISFIABILITY
Instance: A boolean formula φ.
Parameter: A positive integer k.
Question: Is there a set of k variables and a truth table assignment to those variables that causes φ to unravel?

This problem is $W[P]$-complete by a reduction from k-INDUCED 3CNF SATISFIABILITY (Abrahamson, Downey, and Fellows [2]).

CHAIN REACTION CLOSURE
Instance: A directed graph $D = (V, A)$.
Parameter: A positive integer k.
Question: Does there exist a set V' of k vertices of D whose chain reaction closure is D? (A *chain reaction closure of* V' is the smallest superset S of V' such that if $u, u' \in S$ and arcs ux, $u'x$ are in D then $x \in S$.)

This problem is $W[P]$-complete by a reduction from WEIGHTED MONOTONE CIRCUIT SATISFIABILITY (Abrahamson, Downey, and Fellows [2]).

DEGREE 3 SUBGRAPH ANNIHILATOR
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Is there a set of k vertices $V' \subseteq V$ such that $G - V'$ has no subgraph of minimum degree 3?

This problem is $W[P]$-complete by a reduction from WEIGHTED MONOTONE CIRCUIT SATISFIABILITY (Abrahamson, Downey, and Fellows [2]).

LINEAR INEQUALITIES
Instance: A system of linear inequalities.
Parameter: A positive integer k.
Question: Can we delete k of the equalities and get a system that is consistent over the rationals?

This problem is $W[P]$-complete by a reduction from WEIGHTED MONOTONE CIRCUIT SATISFIABILITY taken from Abrahamson et al. [3] (Abrahamson, Downey, and Fellows [2]).

MINIMUM AXIOM SET
Instance: A finite set S of sentences, an implication relation R consisiting of pairs (A, t) where $A \subseteq S$ and $t \in S$.
Parameter: A positive integer k.
Question: Is there a set $S_0 \subseteq S$ with $|S_0| \leq k$ and a positive integer n such that if we define S_i, $1 \leq i \leq n$, to consist of exactly those $t \in S$ for which either $t \in S_{i-1}$ or there exists a set $U \subseteq S_{i-1}$ such that if $(U, t) \in R$ then $S_n = S$?

The general version of this problem is *NP*-complete by a reduction from X3C (see Garey and Johnson [81, L017]). This problem is $W[P]$-complete by a reduction from WEIGHTED CIRCUIT SATISFIABILITY (Downey et al. [63] and Abrahamson, Downey, and Fellows [2]).

SHORT CIRCUIT SATISFIABILITY
Instance: A boolean circuit C with n gates and at most $k \log n$ inputs and one output.
Parameter: k
Question: Is there a setting of the inputs that cause C to output 1?

This problem is $W[P]$-complete by a reduction from WEIGHTED CIRCUIT SATISFIABILITY (Abrahamson et al. [3], also Abrahamson, Downey, and Fellows [2]).

SHORT SATISFIABILITY
Instance: A formula φ on n variables, a list of at most $k \log n$ variables of φ.
Parameter: k
Question: Is there any setting of the distinguished varaibles that causes φ to unravel?

This problem is $W[P]$-complete by a reduction from SHORT CIRCUIT SATISFIABILITY (Abrahamson, Downey, and Fellows [2]).

THRESHOLD STARTING SET
Instance: A directed graph $D = (V, A)$.
Parameter: A positive integer k.
Question: Does G have a *starting set* of size k? (A *starting set* is a set of vertices $V' \subseteq V$ with the property that if we begin with a pebble on each of the vertices in V' and subsequently place pebbles on any vertex having at least t incoming arcs from pebbled vertices then eventually every vertex of the graph is pebbled.)

This problem is $W[P]$-complete by a reduction from WEIGHTED MONOTONE CIRCUIT SATISFIABILITY (Abrahamson, Downey, and Fellows [2]).

WEIGHTED MONOTONE CIRCUIT SATISFIABILITY
Instance: A boolean monotone circuit C.

Parameter: A positive integer k.
Question: Is there a weight k input vector accepted by C?

This problem is $W[P]$-complete by a reduction from MINIMUM AXIOM SET (Downey et al. [63] and Abrahamson, Downey, and Fellows [2]).

WEIGHTED PLANAR CIRCUIT SATISAFIABILITY
Instance: A planar decision circuit C.
Parameter: A positive integer k.
Question: Does C have a satisfying assignment of weight k?

This problem is $W[P]$-complete by a reduction from WEIGHTED CIRCUIT SATISFIABILITY (Abrahamson, Downey, and Fellows [2]).

Comment. The following classes are not discussed in this paper but are listed for completeness. They are analogues of the QBFSAT and PSPACE in some sense and correspond as we see to the complexity of k-move games. They are discussed at length in [1], [2]. In each case the first problem defines the class.

5.16. *AW[SAT]*-complete.

PARAMETERIZED QBFSAT
Instance: An integer r, a sequence $s_1, \ldots, s_r$ of pairwise disjoint sets of boolean variables, a boolean formula X involving the variables $s_1 \cup \cdots \cup s_r$, integers $k_1, \ldots, k_r$.
Parameter: $k_1, \ldots, k_r$
Question: Is it the case that there exists a size k_1 subset t_1 of s_1 such that for every size k_2 subset t_2 of s_2 there exists a size k_3 subset t_3 of s_3 such that ... (alternating quantifiers) such that, when the variables in $t_1 \cup \cdots \cup t_r$ are made true and all other variables are made false, formula X is true?

PARAMETERIZED MONOTONE QBFSAT

This is the same as PARAMETERIZED QBFSAT except that the formulae are monotone (Abrahamson, Downey, and Fellows [2]).

PARAMETERIZED ANTIMONOTONE QBFSAT

This is the same as PARAMETERIZED QBFSAT except that the formulae are antimonotome (Abrahamson, Downey, and Fellows [2]).

5.17. *AW[SAT]*-hard.

COMPACT DTM COMPUTATION I
Instance: A deterministic Turing machine M operating on tape alphabet Σ, a word $x \in \Sigma^*$.
Parameter: A positive integer k.
Question: Does M on input x accept after visiting at most k work tape squares?

This problem is $AW[SAT]$-hard by a reduction from PARAMETERIZED QBFSAT (r, $k_1, \ldots, k_r$) (Cesati [44]).

COMPACT DTM COMPUTATION II
Instance: A deterministic Turing machine M operating on tape alphabet Σ, a word $x \in \Sigma^*$.
Parameter: A positive integer k, $|x|$.
Question: Does M on input x accept after visiting at most k work tape squares?

This problem is $AW[SAT]$-hard by a reduction from PARAMETERIZED QBFSAT (r, $k_1, \ldots, k_r$) (Cesati [44]).

5.18. *AW[P]*-complete.

PARAMETERIZED QCSAT
Instance: An integer r, a sequence $s_1, \ldots, s_r$ of pairwise disjoint sets of boolean variables, a circuit X with the variables $s_1 \cup \cdots \cup s_r$ as inputs, integers $k_1, \ldots, k_r$.
Parameter: $r, k_1, \ldots, k_r$.
Question: Is it the case that there exists a size k_1 subset t_1 of s_1 such that for every size k_2 subset t_2 of s_2 there exists a size k_3 subset t_3 of s_3 such that ... (alternating quantifiers) such that, when the inputs in $t_1 \cup \cdots \cup t_r$ are set to 1 and all other inputs are set to 0, circuit X outputs 1?

PARAMETERIZED MONOTONE QCSAT
Instance: An integer r, a sequence $s_1, \ldots, s_r$ of pairwise disjoint sets of boolean variables, a montone circuit X with the variables $s_1 \cup \cdots \cup s_r$ as inputs, integers $k_1, \ldots, k_r$.
Parameter: $r, k_1, \ldots, k_r$.
Question: Is it the case that there exists a size k_1 subset t_1 of s_1 such that for every size k_2 subset t_2 of s_2 there exists a size k_3 subset t_3 of s_3 such that ... (alternating quantifiers) such that, when the inputs in $t_1 \cup \cdots \cup t_r$ are set to 1 and all other inputs are set to 0, circuit X outputs 1?

This problem is $AW[P]$-complete by a reduction from PARAMETERIZED QCSAT (Abrahamson, Downey, Fellows, [2]) and MINIMUM AXIOM SET (Downey et al. [63]).

PARAMETERIZED ANTIMONOTONE QCSAT

This is the same as PARAMETERIZED QCSAT except the circuit must be antimonotone. $AW[P]$ complete by a reduction from PARAMETERIZED MONOTONE QCSAT. (Abrahamson, Downey, and Fellows [2]).

5.19. *AW[P]*-hard.

COMPACT TURING MACHINE COMPUTATION
Instance: A nondeterministic Turing machine M operating on tape alphabet Σ, a word $x \in \Sigma^*$.
Parameter: A positive integer k.
Question: Is there an accepting computation of M on input x that visits at most k work tape squares?

This problem is $AW[P]$-hard by a reduction from PARAMETERIZED QCSAT $(r, k_1, \ldots, k_r)$ (Abrahamson, Downey, and Fellows [2]).

COMPACT TURING MACHINE COMPUTATION II
Instance: A nondeterministic Turing machine M operating on tape alphabet Σ, a word $x \in \Sigma^*$.
Parameter: A positive integer k, $|x|$.
Question: Is there an accepting computation of M on input x that visits at most k work tape squares?

This problem is $AW[P]$-hard by a reduction from PARAMETERIZED QCSAT $(r, k_1, \ldots, k_r)$ (Abrahamson, Downey, and Fellows [2]).

5.20. *AW[*]=AW[1] =AW[t]*-complete. Note that Abrahamson, Downey, and Fellows proved that $AW[*] = AW[2]$ in [1], [2]. This was recently improved to show that $AW[*] = AW[1]$ (and in fact $N[*] = N[1]$) by Downey, Fellows, and Regan [64].

PARAMETERIZED QBFSAT$_t$
Instance: An integer r, a sequence $s_1, \ldots, s_r$ of pairwise disjoint sets of boolean variables, a boolean formula X involving the variables $s_1 \cup \cdots \cup s_r$ which consists of t alternating layers of conjunctions and disjunctions with negations applied only to variables, integers $k_1, \ldots, k_r$.
Parameter: $r, k_1, \ldots, k_r$.
Question: Is it the case that there exists a size k_1 subset t_1 of s_1 such that for every size k_2 subset t_2 of s_2 there exists a size k_3 subset t_3 of s_3 such that ... (alternating quantifiers) such that, when the variables in $t_1 \cup \cdots \cup t_r$ are made true and all other variables are made false, formula X is true?

UNITARY PARAMETERIZED QBFSAT$_t$
Instance: An integer r, a sequence $s_1, \ldots, s_r$ of pairwise disjoint sets of boolean variables, a boolean formula X involving the variables $s_1 \cup \cdots \cup s_r$ which consists of t alternating layers of conjunctions and disjunctions with negations applied only to variables.
Parameter: A positive integer k.
Question: Is it the case that there exists a variable t_1 of s_1 such that for every variable t_2 of s_2 there exists a variable t_3 of s_3 such that ... (alternating quantifiers) such that, when the variables in $t_1, \ldots, t_r$ are made true and all other variables are made false, formula X is true?

This problem is $AW[t]$-complete by Abrahamson, Downey, and Fellows [1], [2].

SHORT GENERALIZED GEOGRAPHY
Instance: A directed graph $D = (V, A)$, a specified vertex $v_0 \in V$.
Parameter: A positive integer k.
Question: Does player one have a winning strategy in k moves for the following game? Players alternately choose a new arc from A. The first arc chosen must have its tail at v_0 and each subsequently chosen arc must have its tail at the vertex that was the head of the previous arc. The first player unable to choose a new arc loses.

The general version of this problem is $PSPACE$-complete by a reduction from QBFSAT (see Garey and Johnson [81, GP2]). This problem is $AW[*]$-complete by a reduction from UNITARY PARAMETERIZED QBFSAT$_t$ (Abrahamson, Downey, and Fellows [2]).

SHORT NODE KAYLES
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Does player one have a winning k-move strategy in the following game? That is, players pebble a vertex not adjacent to any pebbled vertex. The first player with no play loses. Player one plays first.

The general version of this problem is $PSPACE$-complete by a reduction from QBFSAT (see Garey and Johnson [81, GP3]). This problem is $AW[*]$-complete by a reduction from UNITARY PARAMETERIZED QBFSAT$_t$ (Abrahamson, Downey, and Fellows [2]).

5.21. In W[1].

IRREDUNDANT SET
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Is there a set $V' \subseteq V$ of cardinality k having the property that each vertex $u \in V'$ has a private neighbor? (A *private neighbor* of a vertex $u \in V'$ is a vertex u' (possibly $u' = u$) with the property that for every vertex $v \in V'$, $u \neq v$, $u' \notin N[v]$.)

IRREDUNDANT SET is in $W[1]$ by Downey and Fellows [55].

5.22. Open problems.

BOUNDED HAMMING WEIGHT DISCRETE LOGARITHM
Instance: An n-bit prime, a generator g of F_p^*, an element $a \in F_p^*$.
Parameter: A positive integer k.
Question: Is there a positive integer x whose binary representation has at most k 1's (that is, x has a Hamming weight of k) such that $a = g$?

Candidate for membership in randomized FPT (Fellows and Koblitz [72], [73]). This problem is of practical significance because the use of exponents of fairly small Hamming weight has been suggested in order to speed up cryptosystems based on discrete log (see [72], [73], and references).

DIRECTED FEEDBACK VERTEX SET
Instance: A directed graph $D = (V, A)$.
Parameter: A positive integer k.
Question: Is there a set S of k vertices such that each directed cycle of G contains a member of S?

The general version of this problem is *NP*-complete by a reduction from VERTEX COVER (see Garey and Johnson [81, GT8]). This can be solved in $O(n^{k+1})$ by brute force for each fixed k. A related problem DIRECTED FEEDBACK ARC SET asks for a set A of at most k arcs such that every directed cycle contains at least one arc from A. These problems can be shown to have the same $\leq_m^s$-degree. The undirected version of this problem is in FPT.

IMMERSION ORDER TEST
Instance: A graph $G = (V, E)$.
Parameter: A graph $H = (V', E')$.
Question: Is $H \leq_i G$ where $\leq_i$ denotes the immersion ordering?

JUMP NUMBER
Instance: A poset $P = (P, \leq)$.
Parameter: A positive integer k.
Question: Is the jump number of $P \leq k$?

The general version of this problem is *NP*-complete (See Pulleybank [103]). By El-Zahar and Schmerl [67], there is an $O(n^{k+1})$ algorithm.

PLANAR t-NORMALIZED WEIGHTED SATISFIABILITY
Instance: A planar t–normalized formula X.
Parameter: A positive integer k.
Question: Does X have a satisfying assignment of weight k?

This question is of some interest for $t = 1$ since it might be a candidate for an intractable problem that is not $W[1]$ hard.

PLANAR MULTIWAY CUT
Instance: A weighted planar graph $G = (V, E)$ with terminals $\{x_1, \ldots, x_k\}$.
Parameter: k
Question: Is there a set of edges of total weight $\leq k'$ whose removal disconnects each terminal from all the others?

The general version of this problem is *NP*-complete (Dalhaus et al. [50]). Best known complexity is $O((4^k)^k n^{2k-1} \log n)$ by [50] where it is asked if the problem is FPT.

POLYMATROID RECOGNITION
Instance: A k–polymatroid M.
Parameter: A positive integer k.
Question: Is M hypergraphic?

See Vertigan and Whittle [116].

RESTRICTED VALENCE ISOMORPHISM
Instance: Two graphs $G = (V, E)$ and $H = (V', E')$.
Parameter: A positive integer k.
Question: Are G and H isomorphic graphs such that the valences of the vertices of both G and H are bounded by k?

Luks [97] has shown that there is an $O(n^{f(k)})$ algorithm to decide the parameterized version. The question is whether the problem is FPT. If this problem is $W[1]$-hard then GRAPH ISOMORPHISM is not in P unless the W-Hierarchy collapses. The reader should note that this may give an ingress into the GRAPH ISOMORPHISM problem, in the sense one might be able to demonstrate intractability (i.e., assuming that $W[1] \neq FPT$) of the *un*parameterized version by considering some parameterized version instead. The point is that we have already seen many instances where the "complexity" of the unparameterized version is quite different than the parameterized one.

SHORT CHEAP TOUR
Instance: A graph $G = (V, E)$, an edge weighting $w : E \to \mathbf{Z}$.
Parameter: A positive integer k.
Question: Is there a tour through at least k nodes of G of cost at most S?

The general version of this problem is *NP*-complete by a reduction from HAMILTON CIRCUIT (see Garey and Johnson [81, ND22]). Known to be hard for $W[1]$ if we ask that the tour cost *exactly* S (Downey and Fellows [56]).

SHORT GENERALIZED HEX
Instance: A graph $G = (V, E)$ with two distinguished vertices v_1 and v_2.
Parameter: A positive integer k.
Question: Does player one have a winning strategy of at most k moves in the following game? Player one plays with white pebbles and player two with black ones. Pebbles are placed on nondistinguished vertices alternately by player one then player two. Player one wins if he can construct a path of white vertices from v_1 to v_2.

The general version of this problem is $PSPACE$-complete by a reduction from QBF (see Garey and Johnson [81, GP1]). Candidate for $AW[*]$-completeness (Abrahamson, Downey, and Fellows [2]).

SMALL MINIMUM DEGREE 4 SUBGRAPH
Instance: A graph $G = (V, E)$.
Parameter: A positive integer k.
Question: Is there a subgraph of G of minimum degree at least 4 and of cardinality at most k?

TOPOLOGICAL CONTAINMENT
Instance: A graph $G = (V, E)$.
Parameter: A graph H.
Question: Is H topologically contained in G?

This can be solved in $O(n^{O(|E'|)})$ time by brute force together with the k-DISJOINT PATHS algorithm of Robertson and Seymour [106].

WEIGHTED MONOTONE PLANAR BOOLEAN CIRCUIT SATISFIABILITY
Instance: A monotone planar decision circuit C.
Parameter: A positive integer k.
Question: Does C have a satisfying assignment of weight k?

Candidate for $W[SAT]$-completeness (Abrahamson, Downey, and Fellows [2]).

Acknowledgments. Thanks to Karl Abrahamson for useful early discussions about this work, and for suggestions on improving the exposition. Special thanks to Mike Hallett and H. Todd Wareham, who prepared the final version of the appendix.

REFERENCES

[1] K. A. ABRAHAMSON, R. G. DOWNEY, AND M. R. FELLOWS, *Fixed-Parameter Intractability* II, in STACS 93, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1993, pp. 374–385.

[2] ———, *Fixed-parameter tractability and completeness* IV: *On completeness for $W[P]$ and $PSPACE$ analogues,* Ann. Pure Appl. Logic, to appear.

[3] K. A. ABRAHAMSON, J. ELLIS, M. R. FELLOWS, AND M. MATA, *On the complexity of fixed-parameter problems*, in 30th Annual Symposium on Foundations of Computer Science, IEEE Press, Piscataway, NJ, 1989, pp. 210–215.

[4] K. A. ABRAHAMSON AND M. R. FELLOWS, *Cutset regularity beats well-quasi-ordering for bounded treewidth*, Tech. Report, Computer Science Dept., Univ. of Idaho, 1989.

[5] ———, *Finite automata, bounded treewidth, and well-quasiordering,* in Graph Structure Theory, N. Robertson and P. Seymour, eds., Contemporary Mathematics Vol. 147, American Mathematical Society, Providence, RI, 1993, pp. 539–564.

[6] R. ANDERSON AND E. MAYR, *A P-complete problem and approximations to it*, Stanford University, Tech. Report STAN-CS-84-1014, 1984.

[7] S. ARNBORG, *Efficient algorithms for combinatorial problems on graphs with bounded decomposability — a survey,* BIT, 25 (1985), pp. 2–23.

[8] S. ARNBORG, D. G. CORNEIL, AND A. PROSKUROWSKI, *Complexity of finding embeddings in a k-tree*, SIAM J. Alg. Discrete Meths., 8, (1987), pp. 277–284.

[9] S. ARNBORG, B. COURCELLE, A. PROSKUROWSKI, AND D. SEESE, *An Algebraic Theory of Graph Reduction*, Tech. Report 90–02, Laboratoire Bordelais de Recherche Informatique, Univ. Bordeaux I, January 1990.

[10] S. ARNBORG, J. LAGERGREN, AND D. SEESE, *Problems easy for tree-decomposible graphs (extended abstract)*, in Proc. 15th Coll. Automata, Languages and Programming, T. Lepisto and A. Salomaa, eds., Lecture Notes in Computer Science, Vol. 317, Springer-Verlag, Berlin, 1988, pp. 38–51.

[11] S. AUSIELLO, A. D'ATRI, AND M. PROTASI, *Structure preserving reductions among convex optimization problems*, J. Comput. System Sci., 21 (1980), pp. 136–153.

[12] J. BALCAZAAR, J. DIAZ, AND J. GABARRO, *Structural Complexity*, Vols. 1 and 2, Springer-Verlag, Berlin, 1987, 1989.

[13] M. W. BERN, E. L. LAWLER, AND A. L. WONG, *Linear time computation of optimal subgraphs of decomposable graphs*, J. Algorithms, 8 (1987), pp. 216–235.

[14] D. BIENSTOCK AND M. A. LANGSTON, *Algorithmic implications of the graph minor theorem*, in Handbook of Operations Research and Management Science: Volume on Networks and Distribution, to appear.

[15] D. BIENSTOCK AND C. MONMA, *On the complexity of covering vertices by faces in planar graphs*, SIAM J. Comput., 17 (1988), pp. 53–76.

[16] H. L. BODLAENDER, *On linear time minor tests and depth–first search*, in Proc. 1st Workshop on Algorithms and Data Structures, F. Dehne et al., eds., Lecture Notes in Computer Science, Vol. 382, Springer-Verlag, Berlin, 1987, pp. 577–590.

[17] ———, *Classes of Graphs with Bounded Tree–Width*, Tech. Report RUU–CS–86–22, Dept. of Computer Science, Univ. of Utrecht, Utrecht, The Netherlands, 1986.

[18] ———, *Dynamic programming on graphs with bounded tree–width*, in Proc. 15th Int. Coll. Automata, Languages and Programming, T. Lepistö and A. Salomaa, eds., Lecture Notes in Computer Science, Vol. 317, Springer-Verlag, Berlin, 1988, pp. 103–118.

[19] ———, *Some classes of graphs with bounded treewidth*, Bull. EATCS, 36 (1988), pp. 116–126.

[20] ———, *On Disjoint Cycles*, Tech. Report RUU-CS-90-29, Dept. of Computer Science, Univ. of Utrecht, Utrecht, The Netherlands, August 1990.

[21] H. L. BODLAENDER, private communication, 1994.
[22] ———, *A Tourist's Guide Through Treewidth*, Tech. Report RUU-CS-92-12, Computer Science Department, Univ. of Utrecht, Utrecht, The Netherlands, March 1992.
[23] ———, *A linear time algorithm for finding tree-decompositions of small treewidth*, in Proceedings of the 25th ACM Symposium on Theory of Computing, ACM Press, 1993, pp. 226–234.
[24] H. L. BODLAENDER, R. G. DOWNEY, M. R. FELLOWS, M. T. HALLETT, AND H. T. WAREHAM, *Parameterized complexity analysis in computational biology*, in The IEEE Workshop on Shape and Pattern Matching in Computational Biology, held in conjunction with 1994 IEEE Conference on Computer Vision and Pattern Recognition, June 1994.
[25] H. L. BODLAENDER, R. G. DOWNEY, M. R. FELLOWS, AND H. T. WAREHAM, *The parameterized complexity of sequence alignment and consensus (extended abstract)*, in Proceedings of the Fifth Conference on Combinatorial Pattern Matching (CPM '94), Springer-Verlag, Lecture Notes in Computer Science 807, 1994, pp. 15–30.
[26] ———, *The parameterized complexity of sequence alignment and consensus*, Theoret. Comput. Sci, to appear.
[27] H. L. BODLAENDER AND J. ENGELFRIET, *Domino Treewidth*, Tech. Report UU-CS-1994-11, Department of Computer Science, University of Utrecht, Utrecht, The Netherlands, 1994.
[28] H. L. BODLAENDER, M. R. FELLOWS, AND M. T. HALLETT, *Beyond NP-completeness for problems of bounded width: Hardness for the W-hierarchy (extended abstract)*, STOC 26, 1994, pp. 449–458.
[29] H. L. BODLAENDER, M. R. FELLOWS, AND T. WARNOW, *Two Strikes Against Perfect Phylogeny*, Tech. Report RUU-CS-92-08, Department of Computer Science, University of Utrecht, Utrecht, The Netherlands, 1992.
[30] H. L. BODLAENDER AND D. KRATSCH, private communication, 1994.
[31] R. B. BORIE, R. G. PARKER AND C. A. TOVEY, *Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families*, Algorithmica, 7 (1992), pp. 555–582.
[32] G. BRIGHTWELL AND D. WINKLER, *Counting linear extensions*, Order, 8 (1991), pp. 225–242.
[33] ———, *Counting linear extensions is #P–complete*, Proc. 23rd STOC, IEEE Press, Piscataway, NJ, 1991, pp. 175–181.
[34] S. BUSS, private communication, 1989.
[35] J. F. BUSS AND J. GOLDSMITH, *Nondeterminism within P*, SIAM J. Comput., 22 (1993), pp. 560–572.
[36] L. CAI, *Fixed parameter tractability and approximation problems*, Project Report, Computer Science Department, Texas A&M University, June 1992.
[37] L. CAI AND J. CHEN, *On the amount of nondeterminism and the power of verifying*, Lecture Notes in Computer Science, Vol. 711 (MFCS'93), Springer-Verlag, Berlin, 1993, pp. 311–320.
[38] ———, *On Input Read Modes of Alternating Turing Machines*, Tech. Report 93-046, Department of Computer Science, Texas A&M University, College Station, TX, 1993.
[39] ———, *Fixed-parameter tractability and approximability of NP-hard optimization problems*, Proceedings of the 2nd Israel Symposium on Theory of Computing and Systems, 1993, pp. 118–126.
[40] L. CAI, J. CHEN, R. G. DOWNEY, AND M. R. FELLOWS, *The parameterized complexity of short computation and factorization*, Archive Math. Logic, to appear.
[41] ———, *Advice classes of parameterized tractability*, Annals Pure and Applied Logic, to appear.
[42] ———, *On the structure of parameterized problems in NP*, in STACS '94, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1994, pp. 509–520.
[43] K. CATTELL, M. DINNEEN, AND M. R. FELLOWS, *A simple linear-time algorithm for finding path-decompositions of small width*, to appear.
[44] M. CESATI, private communication, 1994.
[45] P. CHOLAK AND R. G. DOWNEY, *Undecidability and definability for parameterized polynomial time m–reducibilities*, in Logical Methods, J. Crossley, J. Remmel, R. Shore and M. Sweedler eds., Birkhauser, Boston, 1994, pp. 194–221.
[46] C. CHEGIREDDY AND H. HAMACHER, *Algorithms for finding the k best perfect matchings*, J. Combin. Theory Ser. B., (1987), pp. 155–165.
[47] A. CONDON, *The complexity of the max word problem and the power of one way interactive proof systems*, in STACS '91, Lecture Notes in Computer Science, Springer-Verlag, 1991.
[48] S. COOK, *The complexity of theorem proving procedures*, Proceedings 3rd STOC, ACM Press, 1971, pp. 151–158.
[49] B. COURCELLE, *Graph rewriting: An algebraic and logic approach*, in Handbook of Theoretical Computer Science, Vol. B, J. van Leeuwen, ed., North–Holland, Amsterdam, 1990.
[50] E. DAHLHAUS, D. JOHNSON, C. PAPADIMITRIOU, P. SEYMOUR, AND M. YANNAKAKIS, *The complexity of multiway cuts*, in STOC '92, ACM Press, 1992, pp. 241–251.

[51] M. DAVIS, *Unsolvable problems*, in The Handbook of Mathematical Logic, J. Barwise, ed., Elsevier, New York, 1977.
[52] W. DAY AND D. SANKOFF, *Computational complexity of inferring phylogenies by compatibility*, Systematic Zoology, 352 (1986), pp. 224–229.
[53] J. DÍAZ AND J. TORÁN, *Classes of bounded nondeterminism*, Math. Systems Theory, 23 (1990), pp. 21–32.
[54] R. G. DOWNEY, P. EVANS, AND M. R. FELLOWS, *Parameterized learning complexity*, in Proceedings of the 6th Annual Conference on Learning Theory, COLT'93, ACM Press, New York, 1993, pp. 51–57.
[55] R. G. DOWNEY AND M. R. FELLOWS, *Fixed-parameter tractability and completeness*, Congr. Numer., 87 (1992), pp. 161–187.
[56] ———, *Fixed-parameter tractability and completeness* II: *On completeness for W*[1], Theoret. Comput. Sci., to appear.
[57] ———, *Fixed parameter intractability (extended abstract)*, in Proc. Structure in Complexity Theory, IEEE Press, Piscataway, NJ, 1992, pp. 36–50.
[58] ———,*Fixed-parameter tractability and completeness* III: *some structural aspects of the W-hierarchy*, in Complexity Theory, K. Ambos-Spies, S. Homer, and U. Schoning, eds., Cambridge University Press, 1993, pp. 166–191.
[59] ———, *Parameterized computational feasibility*, in Proceedings of Feasible Mathematics II, P. Clote and J. B. Remmel, eds., Birkhauser, Boston, 1995, pp. 219–244.
[60] ———, *Fixed-parameter tractability and intractability-a survey*, in preparation.
[61] ———, *Parameterized Complexity*, monograph in preparation.
[62] R. G. DOWNEY, M. R. FELLOWS, B. KAPRON, M. T. HALLETT, AND H. T. WAREHAM, *Parameterized complexity of some problems in logic and linguistics (extended abstract)*, in the 2nd Workshop on Structural Complexity and Recursion-Theoretic Methods in Logic Programming, 1993. Also in Logic at St. Petersberg, A. Nerode and Y. Matiyasevich, eds., Lecture Notes in Computer Science, Vol. 813, 1994, pp. 89–101.
[63] ———, *Parameterized complexity of some problems in logic and linguistics*, in preparation.
[64] R. G. DOWNEY, M. R. FELLOWS AND K. REGAN, *Parameterized circuit complexity and the W-heirarchy*, to appear.
[65] S. DREYFUS AND R. WAGNER, *The Steiner problem in graphs*, Networks, 1 (1971), pp. 195–207.
[66] E. DUBROVA, private communication, 1994.
[67] M. EL–ZAHAR AND J. SCHMERL, *On the size of jump critical ordered sets*, Order, 1 (1984), pp. 3–5.
[68] P. ERDÖS AND L. PÓSA, *On independent circuits contained in a graph*, Canad. J. Math., (1965), pp. 347–352.
[69] M. R. FELLOWS, *The Robertson-Seymour theorems: A survey of applications*, in Contemporary Mathematics, Vol. 89, American Mathematics Society, Providence, RI, 1989, pp. 1–18.
[70] ———, private communication, 1994.
[71] M. R. FELLOWS, M. T. HALLETT, AND H. T. WAREHAM, *DNA physical mapping: Three ways difficult*, in Proceedings: ESA'93 - European Symposium on Algorithms, T. Lengauer ed., Lecture Notes in Computer Science, Vol. 726, Springer-Verlag, Berlin, 1993, pp. 157–168.
[72] M. R. FELLOWS AND N. KOBLITZ, *Self-witnessing polynomial-time complexity and prime factorization*, in Proc. 7th Structure in Complexity, IEEE Press, Piscataway, NJ, 1992, pp. 107–111.
[73] ———, *Fixed-Parameter Complexity and Cryptography*, Tech. Report DCS-207-IR, Department of Computer Science, University of Victoria, Victoria, BC, 1992.
[74] M. R. FELLOWS AND M. A. LANGSTON, *Nonconstructive tools for proving polynomial time decidability*, J. Assoc. Comput. Mach., 35 (1988), pp. 727–739.
[75] ———, *Layout permutation problems and well-partially ordered sets*, in Proc. 5th MIT Conf. on Advanced Research in VLSI, MIT Press, 1988, pp. 315–327.
[76] ———, *On search, decision and the efficiency of polynomial-time algorithms*, in Proc. Symp. on Theory of Computing (STOC), 1989, ACM Press, pp. 501–512.
[77] ———, *An analogue of the Myhill-Nerode theorem and its use in computing finite basis characterizations*, in Proc. Symp. Foundations of Comp. Sci. (FOCS), IEEE Press, 1989, pp. 520–525.
[78] ———, *Constructivity issues in graph algorithms*, in Conf. on Constructivity Issues in Computer Science, San Antonio, TX, 1991, Lecture Notes in Computer Science, Springer-Verlag, 1991.
[79] ———, *On well-partial-ordering theory and its applications to combinatorial problems in VLSI design*, SIAM J. Discrete Math., 5 (1992), pp. 117–126.
[80] ———, *Nonconstructive advances in polynomial-time complexity*, Inform. Process. Lett., 28 (1987/88), pp. 157–162.
[81] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-completeness*, Freeman, San Francisco, 1979.
[82] W. GASARCH, M. KRENTEL, AND K. RAPPOPORT, *OptP as the normal behavior of NP-complete problems*, Math. Systems Theory, to appear.

[83] J. G. GESKE, *On the Structure of Intractable Sets*, Ph.D. thesis, Iowa State University, Ames, IA, 1987.
[84] P. GOLDBERG, M. GOLUMBIC, H. KAPLAN, AND R. SHAMIR, *Three strikes against physical mapping of DNA (extended abstract)*, manuscript, 1994.
[85] M. GOLUMBIC, H. KAPLAN, AND R. SHAMIR, *On the Complexity of DNA Physical Mapping*, Tech. Report 271/93, Tel Aviv University, Tel Aviv, Israel, 1993.
[86] Y. GUREVICH AND S. SHELAH, *Nearly linear time*, in Logic at Botik '89, A. R. Meyer and M. A. Taitslin, eds., Lecture Notes in Computer Science, Vol. 363, Springer-Verlag, Berlin, 1989, pp. 108–118.
[87] M. T. HALLETT, personal communication, 1994.
[88] J. HOPCROFT AND J. ULLMANN, *Introduction to Automata Theory*, Addison-Wesley, Reading, MA, 1979.
[89] A. ITAI, M. RODEH, AND S. TANIMOTO, *Some matching problems in bipartite graphs*, J. Assoc. Comput. Mach., 25 (1978), pp. 517–525.
[90] H. KAPLAN AND R. SHAMIR, *Pathwidth, Bandwidth and Completion Problems to Proper Interval Graphs with Small Cliques*, Tech. Report 285/93, Tel Aviv University, Tel Aviv, Israel, 1993.
[91] H. KAPLAN, R. SHAMIR, AND R. E. TARJAN, *Tractability of parameterized completion problems on chordal and interval graphs: minimum fill-in and DNA physical mapping (extended abstract)*, Proc. 35th Symposium on Foundations of Computer Science (FOCS), IEEE Press, November 1994, pp. 780–791.
[92] R. KARP, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, R. Miller and J. Thatcher, eds., Plenum Press, New York, 1973, pp. 45–68.
[93] C. KINTALA AND P. FISCHER, *Refining nondeterminism in relativised polynomial time bounded computations*, SIAM J. Comput., 9 (1980), pp. 46–53.
[94] R. LADNER, *On the structure of polynomial time reducibility*, J. Assoc. Comput. Mach., 22 (1975), pp. 155–171.
[95] J. LAGERGREN, *Algorithms and minimal forbiddent minors for tree decomposible graphs*, Ph.D. Dissertation, Dept. of Numerical Analysis and Computing Sciences, Royal Institute of Technology, Stockholm, Sweden, March 1991.
[96] H. R. LEWIS, *Complexity results for classes of quantificational formulas*, J. Comput. Systems Sci., 21 (1980), pp. 317–353.
[97] E. LUKS, *Isomorphism of graphs of bounded valence can be tested in polynomial time*, J. Comput. Systems Sci., 25 (1982), pp. 42–65.
[98] A. NAIK, K. REGAN, AND D. SIVAKUMAR, *Quasilinear time complexity theory*, in STACS '94, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1994, pp. 97–108.
[99] J. NESETRIL AND S. POLJAK, *On the complexity of the subgraph problem*, Comm. Math. Univ. Carol., 26 (1985), pp. 415–419.
[100] C. PAPADIMITRIOU AND M. YANNAKAKIS, *On limited nondeterminism and the complexity of the V-C dimension*, Eighth Annual Conference on Structure in Complexity Theory, IEEE Press, 1993 pp. 12–18.
[101] A. PAZ AND S. MORAN, *Nondeterministic polynomial optimization problems and their approximations*, Theoret. Comput. Sci., 15 (1981), pp. 251–277.
[102] J. PLEHN AND B. VOIGT, *Finding minimally weighted subgraphs*, in Proc. 16th International Workshop WG90, Graph-Theoretic Concepts in Computer Science, Lecture Notes in Computer Science 484, Springer-Verlag, (1991), pp. 18–29.
[103] B. PULLEYBANK, *On minimizing setups in precedence constrained scheduling*, Discrete Appl. Math., 11 (1985).
[104] K. REGAN, *Finitary substructure languages*, in Proc. 4th Structure in Complexity Conf., IEEE Press, Piscataway, NJ, 1989, pp. 87–96.
[105] E. RISTAD, *Complexity of Simplified Segmental Phonology*. Tech. Report CS-TR-388-92, Dept. of Computer Science, Princeton University, Princeton, NJ, revised May 1993.
[106] N. ROBERTSON AND P. D. SEYMOUR, *Disjoint paths — a survey*, SIAM J. Discrete Math., 6 (1985), pp. 300–305.
[107] ———, *Graph minors* XIII: *The disjoint paths problem*, to appear.
[108] ———, *Graph minors* XV: *Wagner's conjecture*, to appear.
[109] H. U. SIMON, *Continuous reductions among combinatorial optimization problems*, Acta Informatica, 26 (1989), pp. 771–785.
[110] M. SIPSER, *Borel sets and circuit complexity*, in Proceedings 15th STOC, ACM Press, 1983, pp. 61–69.
[111] W. SLOUGH AND K. WINKLMANN, *On limitations of transformations between combinatorial problems*, Journal Comput. System Sci., to appear.
[112] E. SPILRAJN, *Sur l'extension de l'ordre partiel*, Fund. Math., 16 (1930), pp. 386–389.
[113] L. VALIANT, *The complexity of computing the permanent*, Theoret. Comput. Sci., 8 (1979), pp. 189–201.
[114] L. VALIANT AND V. V. VAZIRANI, *NP is as easy as detecting unique solutions*, Theoret. Comput. Sci., 47 (1986), pp. 85–93.
[115] L. VAN LEEUWEN, *Handbook of Theoretical Computer Science, Vol. A*, North Holland, Amsterdam, 1990.

[116] D. VERTIGAN AND G. WHITTLE, *Recognising polymatroids associated with hypergraphs*, Combin., Probab. Comput., to appear.

[117] H. T. WAREHAM, personal communication, 1994.

[118] T. WIMER, S. HEDETNIEMI, AND R. LASKAR, *A methodology for constructing linear time algorithms*, Congr. Numer., 50 (1985), pp. 43–60.

[119] M. YANNAKAKIS, *Computing the minimum fill-in is NP complete*, SIAM J. Alg. Discrete Meth., 2 (1981), pp. 77–79.

SIAM J. COMPUT.
Vol. 24. No. 5, pp. 923–947, October 1995

COMPUTATIONAL COMPLEXITY OF TWO-DIMENSIONAL REGIONS*

ARTHUR W. CHOU† AND KER-I KO‡

Abstract. The computational complexity of bounded sets of the two-dimensional plane is studied in the discrete computational model. We introduce four notions of polynomial-time computable sets in $\mathbf{R}^2$ and study their relationship. The computational complexity of the winding number problem, membership problem, distance problem, and area problem is characterized by the relations between discrete complexity classes of the NP theory.

Key words. computational complexity, polynomial time, simply closed curves, winding numbers

AMS subject classifications. 68Q05, 68Q15

1. Introduction.

Computational complex analysis studies the algorithmic aspects of the theory of functions of complex variables. While the design and analysis of efficient algorithms plays a central role in the theory, the study of the computational complexity of basic problems in the theory is also an important subject. A noteworthy example is the recent progress on the study of the computational complexity of the fundamental theorem of algebra, including Smale [19], Schönhage [18], and Neff [13] to name just a few.

In this paper, we are concerned with a basic problem in the complexity theory of computational complex analysis: how to measure the complexity of a subset of the plane $\mathbf{R}^2$. In particular, we are interested in the precise notion of polynomial-time computable sets $S \subseteq \mathbf{R}^2$. The notion of polynomial-time computability has played an important role in the development of the discrete complexity theory. We expect that an analogous notion in computational complex analysis is also fundamental in the sense that the complexity of problems in this field can be characterized by this notion and its extensions.

Computational problems involving real-valued functions have been studied in many different settings. Depending upon the motivations and applications, each approach uses a different computational model. In this paper, we are interested in the interplay between real-valued computation and the complexity notions of the discrete NP theory and, therefore, we study these problems using the discrete model of Ko and Friedman [11]. In this model, a real number is represented by a converging sequence of rational numbers and the complexity of an algorithm is measured by the number of bit-operations operated by the algorithm. (See §2 for the formal definitions and discussions on computational models for real-valued computation; see also [9] for a complete treatment.) Following this approach, there are several natural ways to define the notion of polynomial-time computable sets in $\mathbf{R}^2$. We will introduce four such notions and study their relationship.

The first notion is that of polynomial-time approximable sets in $\mathbf{R}^2$. Informally, a subset $S \subseteq \mathbf{R}^2$ is *polynomial-time approximable* if there is a machine M which, on a given point $\mathbf{z} \in \mathbf{R}^2$ and an integer n, determines whether $\mathbf{z}$ is in S within time polynomial in n and admits errors only on a set $E \subseteq \mathbf{R}^2$ of measure $\leq 2^{-n}$. This notion is motivated by the concept of recursively approximable sets developed in recursive analysis, and has been studied in the one-dimensional case in [7]. The second notion is the polynomial-time recognizable sets in $\mathbf{R}^2$. A set $S \subseteq \mathbf{R}^2$ is *polynomial-time recognizable* if there is a machine M which, on a given

*Received by the editors August 27, 1993; accepted for publication March 14, 1994. An extended abstract of this paper appeared in the Proceedings of 25th ACM Symposium on Theory of Computing, 1993.

†Department of Mathematics and Computer Science, Clark University, Worcester, Massachusetts 01610 (achou@vax.clarku.edu).

‡Department of Computer Science, State University of New York at Stony Brook, Stony Brook, New York 11794 (keriko@sbcs.sunysb.edu). Part of this research was done while the second author was visiting the Department of Computer Science of Princeton University. The research of this author was supported in part by National Science Foundation grant CCR 9121472.

point $\mathbf{z} \in \mathbf{R}^2$ and an integer n, determines whether $\mathbf{z}$ is in S within time polynomial in n and admits errors only on points $\mathbf{z}$ that are within a distance 2^{-n} of the *boundary* of S. This definition is more useful for bounded open sets. The relation between the above two notions of polynomial-time computable sets has a close connection to the well-known open question of whether polynomial-time probabilistic computation (on integers) is strictly stronger than polynomial-time deterministic computation (on integers).

Note that the above two definitions allow the machines M to make *two-sided* errors. If we make the further restriction that the machines M can only make *one-sided errors*, so that M must output the correct answer when $\mathbf{z} \in S$, then we obtain *strongly polynomial-time approximable* and *strongly polynomial-time recognizable* sets. It is interesting to point out that the class of strongly polynomial-time recognizable sets precisely characterizes the zeros of polynomial-time computable functions defined on $\mathbf{R}^2$ (with a polynomial inverse modulus at zeros). This characterization allows us to derive complexity bounds for the zeros of polynomial-time computable functions defined on $\mathbf{R}^2$.

A basic object in computational complex analysis is a bounded, simply connected region; that is, a bounded, connected open set with no hole (or, equivalently, whose complement is connected). The boundary curve is a natural representation for such a set. We study some fundamental issues concerning such regions with polynomial-time boundary curve representations. The most critical problem is the *membership problem*: Must a simply connected region with a polynomial-time computable boundary curve be polynomial-time recognizable? This question turns out to be closely related to the *winding number problem* of computing the winding number of a given curve. We show that the complexity of the winding number problem can be characterized by the counting complexity class $\#P$. Namely, if a curve is polynomial-time computable then the winding number problem is solvable in polynomial-time using a function $f \in \#P$ as an oracle; conversely, a polynomial-time computable curve could be designed in such a way that its winding number problem encodes a discrete $\#P$-complete problem. For the membership problem, it immediately follows that it can also be solved in polynomial time relative to an oracle $f \in \#P$. In addition, we can show that it is not necessarily polynomial-time recognizable unless weak one-way functions do not exist (more precisely, $P = UP$). It remains an open question whether the gap between the upper bound $P^{\#P}$ and the lower bound UP could be narrowed.

In addition to the membership problem, we also investigate the *distance problem* of determining the distance between a polynomial-time computable curve and a given point in $\mathbf{R}^2$, and the *area problem* of determining the measure of a simply connected region (represented by a polynomial-time computable boundary curve). The distance problem is easily seen as equivalent to the minimization problem on one-dimensional real functions. We show that the distance problem is polynomial-time solvable if and only if $P = NP$. For the area problem, we show that the area of a region with a rectifiable boundary curve (i.e., a curve of a finite length) must be polynomial-time computable if and only if $FP = \#P$. In contrast, it is interesting to note that if the boundary curve is nonrectifiable, then the area of a region with a polynomial-time computable boundary curve could be noncomputable at all (see [10]).

The above results demonstrate an interesting relation between the continuous problems on two-dimensional regions and the complexity theory of discrete computation. They serve as the first step toward an understanding of the complexity of problems in computational complex analysis.

2. Preliminaries.

2.1. Notation and terminology. This paper deals with both discrete computation on strings in $\{0, 1\}^*$ and continuous computation on real numbers. The basic computational objects in discrete computation are integers and strings. The length of a string w is denoted

by $\ell(w)$. We write $\langle w_1, w_2\rangle$ to denote the pairing function on w_1 and w_2. The complement of a set S is written as S^c.

The basic computational objects in continuous computation are dyadic rationals $\mathbf{D} = \{m/2^n : m \in \mathbf{Z}, n \in \mathbf{N}\}$. Each dyadic rational d has infinitely many binary representations with arbitrarily many trailing zeros. For each such representation s, we write $\ell(s)$ to denote its length. If the specific representation of a dyadic rational d is understood (often the shortest binary representation), then we write $\ell(d)$ to denote the length of this representation. We let $\mathbf{D}_n$ denote the class of dyadic rationals with at most n bits in the fractional part of its binary representation.

For real numbers x and y, we write $\langle x, y\rangle$ to denote a point in the $\mathbf{R}^2$ space. We often write $\mathbf{z}$ to denote a point in $\mathbf{R}^2$. For a real number x, $|x|$ denotes its absolute value. For convenience, we use the L_∞-metric for the space $\mathbf{R}^2$; thus, for two points $\mathbf{z}_1 = \langle x_1, y_1\rangle$, $\mathbf{z}_2 = \langle x_2, y_2\rangle \in \mathbf{R}^2$, $|\mathbf{z}_1 - \mathbf{z}_2|$ denotes their distance $\max\{|x_1 - x_2|, |y_1 - y_2|\}$. For any point $\mathbf{z} \in \mathbf{R}^2$ and any set $S \subseteq \mathbf{R}^2$, we let $\delta(\mathbf{z}, S)$ be the distance between $\mathbf{z}$ and S; i.e., $\delta(\mathbf{z}, S) = \inf\{|\mathbf{z} - \mathbf{y}| : \mathbf{y} \in S\}$. We write $N(\mathbf{z}; \epsilon)$ to denote the open neighborhood of the center $\mathbf{z}$ and radius ϵ; using the L_∞-metric, this is the open square centered at $\mathbf{z}$ and having length 2ϵ at each side. For a set $S \subseteq \mathbf{R}^2$, we let $\mu^*(S)$ be the outer measure of S and $\mu(S)$ be the measure of S if S is measurable.

In this paper, we will consider only bounded subsets of $\mathbf{R}^2$. A subset S of the two-dimensional plane $\mathbf{R}^2$ is called a *region* if it is nonempty, open, and connected. A bounded region is *simply connected* if it does not have "holes" or, equivalently, its complement is also connected. The boundary of a bounded, simply connected region is a simple, continuous, closed curve.

2.2. Complexity classes. The fundamental complexity class we are interested in is the class P of sets acceptable by deterministic polynomial-time (Turing) machines. The following is a list of other important complexity classes we will use in this paper:

NP: the class of sets acceptable by nondeterministic polynomial-time machines,

UP: the class of sets acceptable by unambiguous nondeterministic polynomial-time machines (machines that have at most one accepting computation on any input),

BPP: the class of sets acceptable by polynomial-time probabilistic machines,

FP: the class of functions (mapping strings to strings) computable by deterministic polynomial-time machines, and

#*P*: the class of functions that count the number of accepting computations of nondeterministic polynomial-time machines.

Other nonstandard complexity classes will be defined when they are needed. In addition, relativized complexity classes are also used. The reader is referred to [9] for a review of these complexity classes and their relationship.

2.3. Computational model for real functions. The concept of polynomial-time computable real functions used in this paper was first introduced by Ko and Friedman [11]. This concept is an extension of the notion of recursive real functions used in recursive analysis [16], based on the bit-operation complexity measure defined on Turing machines. This computational model is consistent with the ones used in many other works involving real-valued computation, including [18], [12], and [15]. It is different from the real random access machine (RAM) model used in, e.g., [20] and [2]. Ko [9] contains more discussions on the models for continuous functions.

We give a short summary of the basic notions in this theory. A real number $x \in \mathbf{R}$ is represented by a *Cauchy function* $\phi : \mathbf{N} \to \mathbf{D}$ that binary converges to x in the sense that $\phi(n) \in \mathbf{D}_n$ and $|\phi(n) - x| \leq 2^{-n}$. If the function ϕ further satisfies the condition that $\phi(n) \leq x < \phi(n) + 2^{-n}$, then we call ϕ the *standard Cauchy function* for x and write b_x for ϕ. Another representation for real numbers is the *left cut* representation. For each Cauchy

function ϕ representing x, we let L_ϕ be the set of strings of the form $s.t$ that represent dyadic rationals d such that $d \leq \phi(n)$, where $n = \ell(t)$.[1] The left cut L_{b_x} associated with the standard Cauchy function b_x is called the *standard left cut* of x, and is denoted by L_x.

A real number x is *recursive*, or *computable*, if there exists a computable Cauchy function ϕ that binary converges to x. A Cauchy function $\phi : \mathbf{N} \to \mathbf{D}$ is *polynomial-time computable* if there exist a Turing machine M and a polynomial p such that $M(n)$ computes $\phi(n)$ in time $p(n)$. A real number x is *polynomial-time computable* if there exists a polynomial-time computable Cauchy function that binary converges to x or, equivalently, if there exists a left cut of x in P. (In the following, we will write "P-computable" to mean the term "polynomial-time computable.")

The computational model for real functions is the oracle Turing machine. An oracle Turing machine M is an ordinary Turing machine equipped with an extra query tape and two extra states: the query state and the answer state. The machine M makes a query to an oracle function $\phi : \mathbf{N} \to \mathbf{D}$ as follows: First, it writes an integer k on the query tape, then enters the query tape and waits for the answer $\phi(k)$. The oracle ϕ then reads the input k, replaces the integer k by the dyadic rational number $\phi(k)$, and places the machine M on the answer state. After the machine M enters the answer state, it continues like ordinary machines; in particular, it can read $\phi(k)$ off the query tape. The action from the query state to the answer state counts only one machine step.

We say an oracle Turing machine M is *polynomial time-bounded* if there exists a polynomial p such that for all inputs n and all oracles ϕ, $M^\phi(n)$ halts in time $p(n)$.

DEFINITION 2.1. (a) *A function $f : [0, 1] \to \mathbf{R}$ is* recursive, *or* computable, *if there exists an oracle Turing machine M such that for any oracle function ϕ that binary converges to a real number $x \in [0, 1]$ and any integer n, the machine M outputs a dyadic rational e such that $|e - f(x)| \leq 2^{-n}$. In other words, the oracle machine computes the operator that maps a Cauchy function for x to a Cauchy function for $f(x)$.*

(b) *A function $f : [0, 1] \to \mathbf{R}$ is* P-computable *if it is computable by an oracle Turing machine that operates in polynomial time.*

An equivalent definition for P-computable real functions f will be used in this paper. We say a function $f : [0, 1] \to \mathbf{R}$ has a *polynomial modulus* if there exists a polynomial p such that $|x - y| \leq 2^{-p(n)}$ implies $|f(x) - f(y)| \leq 2^{-n}$.

PROPOSITION 2.2. *A function $f : [0, 1] \to \mathbf{R}$ is* P-computable *iff*

(i) *f has a polynomial modulus, and*

(ii) *there exist a Turing machine M and a polynomial p such that for any dyadic rational number d of length $\leq n$, and any integer n, $M(d, n)$ outputs in time $p(n)$ a dyadic rational number e such that $|e - f(d)| \leq 2^{-n}$.*

The above definition and proposition can be extended to functions mapping $[0, 1]$ to $\mathbf{R}^2$ or functions mapping $[0, 1]^2 \to \mathbf{R}$ in a natural way. That is, the machine computing a function from $[0, 1]$ to $\mathbf{R}^2$ will output two dyadic rationals e_1 and e_2 such that $|\langle e_1, e_2\rangle - f(x)| \leq 2^{-n}$, and the machine computing a function from $[0, 1]^2$ to $\mathbf{R}$ will use two oracles ϕ and ψ, representing two real numbers x and y, such that $|M^{\phi,\psi}(n) - f(\langle x, y\rangle)| \leq 2^{-n}$. (We say (ϕ, ψ) represents the point $\langle x, y\rangle$ in $\mathbf{R}^2$ if ϕ represents x and ψ represents y).

3. Polynomial-time recognizable sets. We are interested in characterizing the class of sets $S \subseteq \mathbf{R}^2$ whose membership problems are solvable in polynomial time. Intuitively, the membership problem of a subset S of $\mathbf{R}^2$ is solvable if there exists a machine M that, for each given point $\mathbf{z} \in \mathbf{R}^2$, determines whether $\mathbf{z}$ is in S, that is, if there is a machine M that

[1] It would be more convenient to define L_ϕ as a set of dyadic rationals. However, since each dyadic rational may have many different representations, each of a different length, the above definition is technically most accurate. The standard left cuts, nevertheless, can be defined in terms of dyadic rationals. That is, $L_x = \{d \in \mathbf{D} : d < x\}$.

computes the characteristic function χ_S of S. In our computational model, the point $\mathbf{z} = \langle x, y \rangle$ is naturally presented to the machine as two oracle functions ϕ and ψ that binary converge to x and y, respectively, and the machine M is an oracle Turing machine. In addition, we note that the oracle machine M cannot solve the membership problem for set S absolutely correctly because, within a finite number of moves, it does not have the ability to distinguish between two close but distinct points in $\mathbf{R}^2$. More precisely, we observe that an oracle machine M always computes a continuous function (Proposition 2.2). Since the only subsets of $\mathbf{R}^2$ whose characteristic functions are continuous are $\emptyset$ and $\mathbf{R}^2$, only these two sets are computable with absolutely no error. Thus, for a nontrivial theory, we must allow the machine M to make errors, while we require that the errors are under control. In this and the next sections, we discuss a few different formulations of polynomial-time solvable subsets of $[0, 1]^2$, and consider their relationship. (We only consider bounded sets S).

First, we consider the following definition, which is a simple generalization of one-dimensional polynomial-time approximable sets of [7] and [9].

DEFINITION 3.1. *A set $S \subseteq \mathbf{R}^2$ is* polynomial-time approximable (P-approximable) *if there exist an oracle Turing machine M and a polynomial p such that for any oracles (ϕ, ψ) representing a point $\mathbf{z} = \langle x, y \rangle \in \mathbf{R}^2$ (i.e., ϕ and ψ binary converge to x and y, respectively) and any input n, M outputs either 0 or 1 in $p(n)$ moves such that the following set $E_n(M)$ has size $\mu^*(E_n(M)) \leq 2^{-n}$:*

$$E_n(M) = \{\mathbf{z} \in \mathbf{R}^2 : \text{there exists } (\phi, \psi) \text{ representing } \mathbf{z} \text{ such that } M^{\phi,\psi}(n) \neq \chi_S(\mathbf{z})\}.$$

When the machine M is understood, we write E_n for $E_n(M)$.

In other words, a set S is P-approximable if we can approximately determine the membership of S with the runtime polynomially dependent on the error size. This appears to be an interesting approach for polynomial-time computable sets; some interesting properties of one-dimensional P-approximable sets have been studied in [7]. In particular, the following characterization of P-approximable sets demonstrate that they can be approximated by simple sets.

DEFINITION 3.2. *A sequence $\{S_n\}$ of finite unions of rectangles is said to be* uniformly polynomial-time computable *if*

(i) *the corners of each rectangle in S_n are in $\mathbf{D}_{p(n)} \times \mathbf{D}_{p(n)}$ for some polynomial p, and*

(ii) *there exists a polynomial-time Turing machine M that, on each input $(\mathbf{d}, n)$, where $\mathbf{d} = \langle d_1, d_2 \rangle \in \mathbf{D} \times \mathbf{D}$, determines whether $\mathbf{d}$ is an interior point of S_n, an exterior point of S_n, or is on the boundary of S_n.*

THEOREM 3.3. *A set $S \subseteq [0, 1] \times [0, 1]$ is P-approximable iff there exists a uniformly polynomial-time computable sequence S_n of finite unions of rectangles such that $\mu^*(S \Delta S_n) \leq 2^{-n}$.*

We first make a fundamental observation on the behavior of oracle machines. Recall that b_x is the standard Cauchy function for x. For $x, y \in [0, 1]$, we write $M^{x,y}(n)$ to denote $M^{b_x, b_y}(n)$.

LEMMA 3.4. *Let M be an oracle machine that P-approximates a set S. Assume that M has a polynomial-time complexity bound p. Let $\mathbf{d} = \langle d_1, d_2 \rangle$ with $d_1, d_2 \in \mathbf{D}_{p(n)}$. Then, $N(\mathbf{d}; 2^{-p(n)}) \subseteq S \cup E_n(M)$ if $M^{d_1,d_2}(n) = 1$, and $N(\mathbf{d}; 2^{-p(n)}) \subseteq S^c \cup E_n(M)$ if $M^{d_1,d_2}(n) = 0$.*[2]

Proof. Assume that $d_1, d_2 \in \mathbf{D}_{p(n)}$ and $M^{d_1,d_2}(n) = 1$. For each $\mathbf{z}$ such that $|\mathbf{z} - \mathbf{d}| < 2^{-p(n)}$, we observe that there exists an oracle representation (ϕ, ψ) for $\mathbf{z}$ such that $\phi(i) = b_{d_1}(i)$ and $\psi(i) = b_{d_2}(i)$ for all $i \leq p(n)$. Thus, the computation of $M^{\phi,\psi}(n)$ works exactly the same as that of $M^{d_1,d_2}(n)$, and hence it outputs 1. That means either $\mathbf{z} \in S$ or $\mathbf{z} \in E_n(M)$.

[2]This lemma also holds for P-recognizable sets as defined in Definition 3.5.

The other case is similar. □

Proof of Theorem 3.3. First assume that S is P-approximable. Let M be an oracle machine that approximates S in time $p(n)$ for some polynomial p. For each point $\mathbf{d} = \langle d_1, d_2\rangle \in \mathbf{D}_{p(n)} \times \mathbf{D}_{p(n)}$, let $R_{\mathbf{d}}$ denote the square $[d_1 - 2^{-p(n)}, d_1 + 2^{-p(n)}] \times [d_2 - 2^{-p(n)}, d_2 + 2^{-p(n)}]$. Define

$$S_n = \bigcup\{R_{\mathbf{d}} : d_1, d_2 \in \mathbf{D}_{p(n)} \cap (0, 1), M^{d_1,d_2}(n) = 1\}.$$

Then, it is easy to see that $\{S_n\}$ is uniformly P-computable as defined in Definition 3.2. We claim that $S_n \Delta S \subseteq E_n(M)$, and hence $\mu^*(S_n \Delta S) \leq 2^{-n}$.

Assume that $\mathbf{z} \in [0, 1]^2$ is in $S_n - S$. Then, there exist $d_1, d_2 \in \mathbf{D}_{p(n)} \cap [0, 1]$ such that $|z - \langle d_1, d_2\rangle| \leq 2^{-p(n)}$ and $M^{d_1,d_2}(n) = 1$. By Lemma 3.4, $\mathbf{z} \in E_n(M)$. Similarly, we can show that if $\mathbf{z}$ is in $S - S_n$, then $\mathbf{z} \in E_n(M)$. So the claim is proven.

Conversely, assume that $\{S_n\}$ is a uniformly P-computable sequence of finite unions of rectangles such that $\mu^*(S_n \Delta S) \leq 2^{-n}$. Let p be a polynomial such that all corners of rectangles in S_n are in $\mathbf{D}_{p(n)} \times \mathbf{D}_{p(n)}$. Let $r(n) = p(n + 1) + n + 2$. Consider the following:

> MACHINE M. With oracles (ϕ, ψ) representing a point $\mathbf{z} \in \mathbf{R}^2$ and on input n, output 1 iff the point $\mathbf{d} = \langle \phi(r(n)), \psi(r(n))\rangle$ is an interior point of S_{n+1}.

Note that if $\mathbf{z}$ has a distance $\geq 2^{-r(n)}$ to the boundary of S_{n+1}, then M must accept $\mathbf{z}$ on input n iff $\mathbf{z} \in S_{n+1}$. Thus, a point $\mathbf{z}$ is in $E_n(M)$ only if $\mathbf{z} \in S_{n+1} \Delta S$ or $\mathbf{z}$ is within a distance $2^{-r(n)}$ to the boundary of S_{n+1}. The boundary of S_{n+1} is contained in the set $B = \{\langle x, y\rangle : x \in \mathbf{D}_{p(n+1)} \cap [0, 1] \text{ or } y \in \mathbf{D}_{p(n+1)} \cap [0, 1]\}$. Since the set of points $\mathbf{z}$ within distance $2^{-r(n)}$ to B is of measure $2^{-r(n)} \cdot 2^{p(n+1)+1} = 2^{-(n+1)}$, the total error $E_n(M)$ has a measure $\leq 2^{-n}$. □

Although the notion of P-approximable sets appears interesting from the measure-theoretic point of view, it does not provide strong control over where the errors may occur. That is, two sets S_1 and S_2 are essentially regarded as the same set here if they differ by at most a set of measure 0. While a set of measure 0 is negligible from the measure-theoretic point of view, it could be an important factor in other computational problems, for instance, the problem of determining the complexity of isolated zeros of a function. The following definition further controls the errors by requiring that errors occur only close to the "boundary" of the set. For any set $S \subseteq \mathbf{R}^2$, we let Γ_S be the set of all points $\mathbf{z}$ in $\mathbf{R}^2$ such that for any $r > 0$, the neighborhood $N(\mathbf{z}; r) = \{\mathbf{y} \in \mathbf{R}^2 : |\mathbf{z} - \mathbf{y}| < r\}$ intersects both S and S^c. For a simply connected region S, Γ_S is its boundary.

DEFINITION 3.5. *A set $S \subseteq \mathbf{R}^2$ is* polynomial-time recognizable (P-recognizable) *if there exist an oracle Turing machine M and a polynomial p such that $M^{\phi,\psi}(n)$ computes $\chi_S(\mathbf{z})$ in time $p(n)$ whenever (ϕ, ψ) represents a point $\mathbf{z}$ whose distance to Γ_S is $> 2^{-n}$; i.e., $E_n(M) \subseteq \{\mathbf{z} : \delta(\mathbf{z}, \Gamma_S) \leq 2^{-n}\}$.*

That is, a set S is P-recognizable if it is approximable by a machine M such that the errors can occur only close to its boundary with the run time of the machine M polynomially dependent on the distance between the given point and the boundary. We intend to use this definition to identify the complexity of simply connected regions. The following example shows that this approach is natural for simple sets. We use the term rectangle to denote a set $[a, b] \times [c, d]$ for some real numbers $a \leq b$, $c \leq d$. We say a rectangle $[a, b] \times [c, d]$ is *degenerate* if $a = b$ or $c = d$.

EXAMPLE 3.6. *A nondegenerate rectangle $[a, b] \times [c, d]$ is P-recognizable iff all real numbers a, b, c, d are polynomial-time computable.*

Proof. The backward direction is obvious. For the forward direction, we assume that $[a, b] \times [c, d]$ is P-recognizable. For convenience, assume that $0 < a < 1 < b$ and $d < -1$,

$1 < c$. Then, for any sufficiently large $n > 0$, we can binary search for a dyadic rational $e \in \mathbf{D}_n \cap [0, 1]$ such that $M^{e,0}(n+1) = 0$ and $M^{e',0}(n+1) = 1$, where $e' = e + 2^{-(n+1)}$. Then, either M is correct on both oracles (b_e, b_0) and $(b_{e'}, b_0)$ and hence $e \leq a \leq e'$, or e or e' is in $E_{n+1}(M)$ and hence it is within distance $2^{-(n+1)}$ of a. In either case, $|e - a| \leq 2^{-n}$. Therefore, the number a is P-computable. The other three numbers b, c, d can be proved to be P-computable in a similar way. □

Although we defined P-recognizable sets by a different error control mechanism, the exact relation between P-approximable sets and P-recognizable sets is not a simple matter. For instance, the above example does not appear to hold for P-approximable sets (see Example 3.10 at the end of this section). In the following we consider their relations.

We first consider the question of whether P-recognizable sets must be P-approximable. Intuitively, if S is a simply connected region with the boundary curve Γ_S, and if Γ_S is a *rectifiable* curve (i.e., Γ_S has a finite length), then the error area of a P-recognizer M for S is small, so S is P-approximable. On the other hand, if Γ_S is not rectifiable, then the error area of M could be very large. We summarize these observations in the following theorem.

THEOREM 3.7. (a) *Let S be a simply connected region in $[0, 1]^2$, whose boundary Γ_S is a simple, closed, rectifiable curve. If S is P-recognizable, then it is also P-approximable.*

(b) *There exists a simply connected region S that is P-recognizable but not P-approximable.*

Proof. (a) Assume that the oracle machine M P-recognizes S as defined in Definition 3.5. Then, for each n, the error set E_n is in the neighborhood of Γ_S of distance 2^{-n}. Since Γ_S has a finite length L, the set E_n has measure $\leq L \cdot 2^{-n}$.

(b) Ko [10] has constructed a P-computable, simple, closed curve Γ that has the following properties: (i) the boundary Γ is not rectifiable; (ii) the measure of the interior S of the curve Γ is not recursive; (iii) the interior S of the curve Γ is P-recognizable. In Theorem 9.1, we will see that the measure of a P-approximable set is P-computable relative to an oracle in $\#P$, and hence is recursive. Thus, the set S is P-recognizable but not P-approximable. □

Next, we consider the question of whether P-approximable sets are always P-recognizable. To do this, we need to introduce the notion of probabilistic computation.

DEFINITION 3.8. *A relation $R \subseteq \{0, 1\}^* \times \{0, 1\}^*$ is* polynomial-length related *if there exists a polynomial p such that $R(s, t)$ implies $\ell(t) = p(\ell(s))$. Two sets $A, B \subseteq \{0, 1\}^*$ are called a BP-*pair *if there exists a polynomial-time predicate R that is polynomial-length related with respect to polynomial p such that*

$$A = \{s \in \{0, 1\}^* : \textit{there exist} \geq (3/4)2^{p(\ell(s))} \textit{ strings } t \textit{ such that } R(s, t)\},$$
$$B = \{s \in \{0, 1\}^* : \textit{there exist} \geq (3/4)2^{p(\ell(s))} \textit{ strings } t \textit{ such that } \neg R(s, t)\}.$$

The above definition of *BP*-pairs is the generalization of the complexity class *BPP*; namely, a set A is in *BPP* iff (A, A^c) is a *BP*-pair. Let A and B be two disjoint subsets of $\{0, 1\}^*$. We say A and B are *P-separable* if there exists a set $C \in P$ such that $A \subseteq C$ and $B \subseteq C^c$. In the following we are going to show that if all *BP*-pairs are P-separable then all P-approximable sets are also P-recognizable. Conversely, if $BPP \neq P$, then P-approximable sets are not necessarily P-recognizable. It is easy to see that the condition that all *BP*-pairs are P-separable implies that $BPP = P$. It is not clear whether the converse holds. The best we know is that if $FP = \#P$, then all *BP*-pairs are P-separable.

THEOREM 3.9. *In the following,* (a) $\Rightarrow$ (b) $\Rightarrow$ (c).

(a) *All BP-pairs (A, B) are P-separable.*

(b) *All P-approximable sets are P-recognizable.*

(c) $BPP = P$.

Proof. (a) $\Rightarrow$ (b). Let S be P-approximable and M_1 be an oracle machine that P-approximates S with run time p. For each $\mathbf{d} = \langle d_1, d_2 \rangle$ with $d_1, d_2 \in \mathbf{D}_n \cap [0, 1]$, define

$N_{\mathbf{d}} = \{\mathbf{e} = \langle e_1, e_2\rangle : e_1, e_2 \in \mathbf{D}_{p(2n)} \cap [0,1], |\mathbf{d} - \mathbf{e}| \leq 2^{-n}\}$. Then, the size $s(N_{\mathbf{d}})$ is greater than $2^{2p(2n)-2n+2}$. We define two sets C_1 and C_0 as follows (in the following, we let $\mathbf{d} = \langle d_1, d_2\rangle$ and $\mathbf{e} = \langle e_1, e_2\rangle$):

$$C_1 = \{\mathbf{d} : d_1, d_2 \in \mathbf{D}_n \cap [0,1], \text{ for } (3/4)s(N_{\mathbf{d}}) \text{ many } \mathbf{e} \in N_{\mathbf{d}}, \ M_1^{e_1,e_2}(2n) = 1\},$$
$$C_0 = \{\mathbf{d} : d_1, d_2 \in \mathbf{D}_n \cap [0,1], \text{ for } (3/4)s(N_{\mathbf{d}}) \text{ many } \mathbf{e} \in N_{\mathbf{d}}, \ M_1^{e_1,e_2}(2n) = 0\}.$$

Then, (C_1, C_0) are a *BP*-pair, and from assumption (a), the pair (C_1, C_0) are P-separable. Let M_2 be a polynomial-time Turing machine that separates C_1 from C_0.

Now we describe an oracle machine M that P-recognizes S as follows.

ALGORITHM M. With oracles (ϕ, ψ) and on input n, first pick $d_1 = \phi(n+2)$ and $d_2 = \psi(n+2)$, and then simulate M_2 on input $\langle d_1, d_2\rangle$.

It is clear that M runs in polynomial time. To see that M indeed recognizes S, we need to show that errors only occur close to the boundary of S or, assuming that the distance of $\mathbf{z}$ to Γ_S is greater than 2^{-n}, we need to show that $\mathbf{z} \notin E_n(M)$. Without loss of generality, assume that $\mathbf{z} \in S$. For any oracles (ϕ, ψ) that represent $\mathbf{z}$, let $d_1 = \phi(n+2)$ and $d_2 = \psi(n+2)$; then the distance between $\mathbf{d} = \langle d_1, d_2\rangle$ and Γ_S is greater than $2^{-(n+1)}$. We claim that M_2 recognizes that $\langle d_1, d_2\rangle \in C_1$, and hence the algorithm M P-recognizes S.

To prove the claim, consider the set $N_{\mathbf{d}}$ of dyadic rational points $\mathbf{e} = \langle e_1, e_2\rangle$ such that $e_1, e_2 \in \mathbf{D}_{p(2n+4)}$ and $|\mathbf{d} - \mathbf{e}| \leq 2^{-(n+2)}$. First, the distance between each $\mathbf{e} \in N_{\mathbf{d}}$ and Γ_S is greater than $2^{-p(2n+4)}$. So, $\mathbf{z} \in S$ implies $\mathbf{e} \in S$ and $N(\mathbf{e}; 2^{-p(2n+4)}) \subseteq S$. Second, if for some $\mathbf{e} \in N_{\mathbf{d}}$ we have $M_1^{e_1,e_2}(2n+4) = 0$, then by Lemma 3.4 the neighborhood $N(\mathbf{e}; 2^{-p(2n+4)})$ is contained in $S^c \cup E_{2n+4}(M_1)$, and hence contained in $E_{2n+4}(M_1)$.

Now observe that the neighborhoods $N(\mathbf{e}; 2^{-p(2n+4)-1})$ are pairwise disjoint, each having the measure $2^{-2p(2n+4)}$. Therefore, by the assumption that M_1 P-approximates S, at most $r = 2^{2p(2n+4)-(2n+4)}$ many $\mathbf{e} \in N_{\mathbf{d}}$ could have $M_1^{e_1,e_2}(2n+4) = 0$. Note that $r \leq 1/4$. Thus, the machine M_2 would correctly output 1 on input $\mathbf{d}$.

(b) $\Rightarrow$ (c). Assume that $BPP \neq P$. We are going to construct a set $S \subseteq [0,1]^2$ that is P-approximable but not P-recognizable. For each string w of length n, let i_w be the integer less than 2^n whose n-bit binary expansion (including leading zeros) is w. Then, let $a_n = 1 - 2^{-(n-1)}$ and $x_w = a_n + i_w \cdot 2^{-2n}$. We note that the interval $[0,1)$ is partitioned into $\{[x_w, x_w + 2^{-2\ell(w)}) : \ell(w) \geq 1\}$. We let $S_w = [x_w, x_w + 2^{-2n}] \times [0,1]$.

Let $A \subseteq \{0,1\}^*$ be a set in $BPP - P$. Then, define $S = \bigcup_{w \in A, \ell(w) \geq 1} S_w$. We claim that S satisfies our needs. First, to see that S is not P-recognizable, we note that $\mathbf{z}_w = \langle x_w + 2^{-(2n+1)}, 1/2\rangle$ has a distance $2^{-(2n+1)}$ away from Γ_S. Thus, if S were P-recognizable, then we could correctly determine whether $\mathbf{z}_w \in S$ in time $p(n)$ for some polynomial p (by simulating the machine that P-recognizes S with the standard oracles of $\mathbf{z}_w$ and the input $2n+2$). This in turn would allow us to determine whether $w \in A$ in time $p(n)$, contradicting the assumption that $A \notin P$.

Next, we show that S is P-approximable. Since $A \in$ BPP, there exist a polynomial time predicate R and a polynomial p such that for all $w \in \{0,1\}^*$ with $\ell(w) = n$,

$$\Pr_{\ell(u)=p(n)}[w \in A \iff R(w,u)] \geq 3/4.$$

That is, a random R-test for $w \in A$ has error probability at most $1/4$. We can reduce this error probability to 2^{-k} by repeating the R-test $c \cdot k$ times and taking the majority answer, where c is a constant (see, for instance, [1]). More precisely, for each string u of length $m = c \cdot k \cdot p(n)$, write $u = u_1 \ldots u_{ck}$ with each u_i having length $|u_i| = p(n)$, and define $Q_k(w,u)$ as the predicate which states that the size of $\{i : 1 \leq i \leq ck, R(w,u_i)\}$ is $> ck/2$. Then, we have

$$\Pr_{\ell(u)=m}[w \in A \iff Q_k(w,u)] \geq 1 - 2^{-k}.$$

Note that Q_k is uniformly P-computable in the sense that there is a polynomial-time Turing machine M_Q that, on input $\langle k, w, v\rangle$, determines whether $Q_k(w, v)$ holds.

For each integer k, divide each S_w with $\ell(w) = n$ into 2^n rectangles $T_{k,w,i}$, $1 \le i \le 2^n$; that is, $T_{k,w,i} = [x_w, x_w + 2^{-2n}] \times [(i-1)2^{-n}, i \cdot 2^{-n}]$. For each i, $1 \le i \le 2^n$, let v_i be the n-bit binary representation for integer i. We define $T_{k,w} = \bigcup\{T_{k,w,i} : Q_k(w, v_i)\}$ and $T_k = \bigcup_{\ell(w)\ge 1} T_{k,w}$. We observe, from the estimation of the error probability for Q-tests, that for each w, $\mu((S \cap S_w)\Delta T_{k,w}) \le 2^{-k} \cdot \mu(S_w)$. This implies that $\mu(S\Delta T_k) \le 2^{-k}$.

Since $Q_k(w, v)$ is uniformly P-computable, we can see that T_k is uniformly P-approximable in the following sense: there is a polynomial-time Turing machine M_1 such that $\mu^*\{\mathbf{z} :$ there exist (ϕ, ψ) representing $\mathbf{z}$ such that $M_1^{(\phi,\psi)}(k) \ne \chi_{T_k}(\mathbf{z})\} \le 2^{-k}$. From the above relation between T_k and S and the machine M_1, we obtain the following machine M for set S.

ALGORITHM M. With oracles (ϕ, ψ) and input k, simulate $M_1^{\phi,\psi}(k+1)$.

It is easy to verify that machine M indeed P-approximates S: the error set $E_k(M) \subseteq E_{k+1}(M_1) \cup (S\Delta T_{k+1})$ has measure

$$\mu(E_k(M)) \le \mu(E_{k+1}(M_1)) + \mu(S\Delta T_{k+1}) \le 2^{-(k+1)} + 2^{-(k+1)} = 2^{-k}. \qquad \square$$

The following is a specific example showing the difference between P-approximable sets and P-recognizable sets. It further demonstrates that P-recognizability is the more natural approach for simply connected regions. Recall that a *tally set* is a set over a singleton alphabet $\{0\}$.

EXAMPLE 3.10. *If there exists a tally set in BPP − P, then there exists a nondegenerate rectangle $[a, b] \times [c, d]$ that is P-approximable but not P-recognizable.*

Sketch of Proof. In Lemma 3.14 of [9], it is shown that for any tally set T, there is a real number $x \in [0, 1]$ such that its standard left cut L_x is P-computable relative to the oracle T, and T is P-computable relative to any left cut L_ϕ of x. Thus, for a tally set $T \in BPP - P$, we obtain a real number $x \in [0, 1]$ whose standard left cut L_x is in BPP, but x is not P-computable. Consider the rectangle $A = [0, x] \times [0, 1]$. We see from Example 3.6 that A is not P-recognizable. We verify that A is P-approximable.

Let M be a polynomial-time *deterministic* Turing machine and p be a polynomial such that for all $d \in \mathbf{D}_n$

$$\Pr_{\ell(w)=p(n)}[M(d, w) = 1 \iff d < x] > 1 - 2^{-(n+1)}.$$

Let $0.w$ denote the dyadic rational whose binary expansion is $0.w$. For each $d \in \mathbf{D}_n \cap [0, 1]$, define

$$I_{d,w} = [d + (0.w) \cdot 2^{-n}, d + (0.w) \cdot 2^{-n} + 2^{-(p(n)+n)}],$$

i.e., if $d = 0.s$, then $I_{d,w} = [0.sw, 0.sw + 2^{-(p(n)+n)}]$. Let

$$S_n = \bigcup\{I_{d,w} \times [0, 1] : d \in \mathbf{D}_n \cap [0, 1], M(d, w) = 1\}.$$

Then, it is not hard to verify that $\{S_n\}$ satisfies the condition of Definition 3.2 and $\mu^*(S_n \Delta S) \le 2^{-n}$. $\square$

4. Strongly P-recognizable sets. The notions of P-approximability and P-recognizability introduced in the last section allow two-sided errors to occur in the computation. We can also extend them to have only one-sided errors; that is, machine M must recognize x correctly if $x \in S$. The one-sided error requirement is useful when we are concerned with simple sets

such as sets of measure 0. (It is clear that a set S of measure 0 is trivially P-approximable. It is also trivially P-recognizable, since a set of measure 0 does not contain a rectangle and hence every point in the set is also a boundary point.)

DEFINITION 4.1. *A set* $S \subseteq \mathbf{R}^2$ *is* strongly P-approximable (strongly P-recognizable) *if* S *is P-approximable* (*P-recognizable, respectively*) *by an oracle machine* M *such that* $M^{\phi,\psi}(n) = 1$ *whenever* (ϕ, ψ) *represents a point* x *in* S.

Strongly P-recognizable sets are interesting because they can be used to characterize the sets of zeros of P-computable real functions with polynomial inverse moduli at zeros. We will prove this result later.

It is clear that strongly P-approximable sets are a proper subclass of P-approximable sets.

THEOREM 4.2. *There exists a P-approximable set that is not strongly P-approximable.*

Proof. Consider the set $S = (\mathbf{D} \cap [0, 1]) \times [0, 1]$. This set has measure 0 and so is P-approximable. On the other hand, for the sake of contradiction, let us suppose that M strongly P-approximates S. Then, for any oracles (ϕ, ψ) representing a point $\mathbf{z}$ in $\mathbf{R}^2$ and any input n, M must halt in $p(n)$ moves for some polynomial p. Let $d_1 = \phi(p(n))$ and $d_2 = \psi(p(n))$. We observe that the machine M cannot distinguish the oracles (ϕ, ψ) from the standard oracles (b_{d_1}, b_{d_2}) for $\langle d_1, d_2\rangle \in S$. Since M strongly p-approximates S, it must accept with the oracles (b_{d_1}, b_{d_2}), and hence it must accept with the oracles (ϕ, ψ). Thus, M must output 1 for all oracles representing any point in $[0, 1]^2$, and hence $\mu(E_n) = 1$, leading to a contradiction. □

For the relation between P-recognizable sets and strongly P-recognizable sets, we first note that a nondegenerate rectangle is strongly P-recognizable iff it is P-recognizable.

EXAMPLE 4.3. *If* a, b, c, d *are P-computable real numbers such that* $a < b$ *and* $c < d$, *then the rectangle* $R = [a, b] \times [c, d]$ *is strongly P-recognizable. Therefore, a nondegenerate rectangle is P-recognizable iff it is strongly P-recognizable.*

Proof. For any integer $n > 0$, consider the rectangle $R_n = [a - 2^{-n}, b + 2^{-n}] \times [c - 2^{-n}, d + 2^{-n}]$. Then, by Example 3.6, R_n is P-recognizable. Thus, we have a machine M_n to recognize R_n such that the only errors of M_n occur within the distance 2^{-n} of the boundary of R_n. Treating M_n as a recognizer for R, it recognizes all points in R correctly and the errors only occur within the distance $2^{-(n-1)}$ of the boundary of R. Now we observe that the machines M_n are uniformly polynomial-time bounded and hence can be combined into a single polynomial-time machine M for R. □

The above result does not hold if the rectangle is a degenerate one, since a degenerate rectangle is of measure 0. Also, the strategy of binary search cannot apply since it is of measure 0. We first consider the complexity of a strongly P-recognizable singleton set.

DEFINITION 4.4. (a) *A real number* x *is NP-*computable *if there exist a nondeterministic Turing machine* M *and a polynomial* p *such that for each input* n,

(i) *at least one computation path of* $M(n)$ *halts in time* $p(n)$, *and*

(ii) *if a computation path of* $M(n)$ *halts with output* $d \in \mathbf{D}$, *then* $|d - x| \leq 2^{-n}$.

(b) *A point* $\mathbf{z} = \langle x, y\rangle \in \mathbf{R}^2$ *is NP-*computable *if both* x *and* y *are NP-computable real numbers.*

In the study of the maximization problems on continuous functions, the following notions of left *NP* real numbers and right *NP* real numbers have been defined [6], [9]. A real number x is a *left NP real number* if there exists a left cut L_ϕ of x such that L_ϕ is in *NP*, and it is a *right NP real number* if there exists a left cut L_ϕ of x such that L_ϕ is in *coNP*. We can characterize the class of *NP*-computable real numbers in terms of these notions.

PROPOSITION 4.5. *A real number* x *is NP-computable iff it is both left NP and right NP.*

Sketch of Proof. If x is *NP*-computable by the nondeterministic Turing machine M, for each $n > 0$, let d_n be the greatest dyadic rational in $\mathbf{D}_n$ such that a computation of $M(n)$

outputs d_n. Then, it is clear that $\phi(n) = d_n$ is a Cauchy function for x and the left cut L_ϕ is in *NP*. Similarly, if we let $\psi(n)$ be the least dyadic rational in $\mathbf{D}_n$ such that a computation of $M(n)$ outputs it, then the left cut L_ψ is in *coNP*.

Conversely, if x has a left cut L_1 in *NP* and a left cut L_2 in *coNP*, then for each $n > 0$, we can nondeterministically guess a dyadic rational d in $\mathbf{D}_n$ such that $d \in L_1$ and $d + 2^{-(n-1)} \in L_2^c$ (from the definition of left cuts, such a point must exist). Then, we must have $|d - x| \leq 2^{-(n-1)}$. □

Remark. It is not clear whether an *NP*-computable real number x must have a left cut in $NP \cap coNP$. Although x must have a left cut in *NP* and a left cut in *coNP*, the two left cuts are necessarily identical. The general question of whether an *NP*-computable real number is always P-computable also remains open. We discuss this question and other related problems in §5 (see also §8).

In the following, we show that if a singleton set $\{\mathbf{z}\}$ is strongly P-recognizable, then $\mathbf{z}$ is *NP*-computable. Whether the converse holds or not is an open question.

THEOREM 4.6. *If* $\{\mathbf{z}\}$ *is strongly P-recognizable, then* $\mathbf{z}$ *is NP-computable.*

Proof. Let M_1 be an oracle machine that strongly P-recognizes $\{\mathbf{z}\}$. Then, for any input n, if $M_1^{\phi,\psi}(n) = 1$, then we know that (ϕ, ψ) represents a point $\mathbf{y}$ within a distance 2^{-n} of $\mathbf{z}$. In particular, $\mathbf{d} = \langle \phi(n), \psi(n) \rangle$ is within distance $2^{-(n-1)}$ of $\mathbf{z}$. So, the following nondeterministic machine M computes $\mathbf{z}$ in polynomial time: on input n, guess a dyadic point $\mathbf{d} = \langle d_1, d_2 \rangle$ with $d_1, d_2 \in \mathbf{D}_{n+1}$, and then simulate $M_1^{d_1,d_2}(n+1)$. It outputs $\mathbf{d}$ if the above simulation accepts, and it does not halt if the simulation rejects. □

COROLLARY 4.7. *There exists a P-recognizable set that is not strongly P-recognizable.*

Proof. It is clear that any singleton set $\{\mathbf{z}\}$ is P-recognizable but only those of *NP*-computable $\mathbf{z}$ are strongly P-recognizable. (Apparently, not every real number is *NP*-computable.) □

Finally, we consider the relation between strongly P-approximable sets and strongly P-recognizable sets.

THEOREM 4.8. *Strongly P-recognizable sets and strongly P-approximable sets are incomparable.*

Proof. First recall the proof of Theorem 4.2. We proved that the set $S = (\mathbf{D} \cap [0, 1]) \times [0, 1]$ is not strongly P-approximable. However, it is strongly P-recognizable: we can just accept every point in $[0, 1] \times [0, 1]$, since all of these points are in Γ_S.

Next, we construct a set S that is strongly P-approximable but not strongly P-recognizable. Let $A \subseteq \mathbf{N}$ be a nonrecursive set and $S = \{\langle 2^{-n}, 0 \rangle : n \in A\}$. It is easy to see that S is strongly P-approximable: we only need to accept a point $\mathbf{z} = \langle x, y \rangle$ iff $x \in [0, 1]$ and $|y| \leq 2^{-(k+1)}$. This machine has errors $E_k \subseteq \{\langle x, y \rangle : x \in [0, 1], |y| \leq 2^{-(k+1)}\}$ and so $\mu(E_k) \leq 2^{-k}$.

On the other hand, if S were strongly P-recognizable, then we could decide whether $n \in A$. More precisely, assume that oracle machine M strongly P-recognizes S. Then, we can see that for any integer n, $n \in A$ iff $M^{\mathbf{z}_n}(n+2) = 1$, where $\mathbf{z}_n = \langle 2^{-n}, 0 \rangle$. This is true because for any $n \notin A$, the distance between $\mathbf{z}_n$ and Γ_S is at least $2^{-(n+1)}$, and so the machine M cannot lie. Also, if $n \in A$, then by the requirement of strong P-recognizability of M, M must output 1 correctly. Thus, the existence of such an oracle machine contradicts the assumption that A is nonrecursive. □

5. Zeros of polynomial-time computable functions. It is known that the class of recursively closed sets precisely characterizes the sets of the zeros of P-computable real functions from $[0, 1]$ to $\mathbf{R}$ [14], [9]. In this section, we show an analogous result for P-computable real functions which have polynomial inverse moduli of continuity at zeros; that is, the sets of zeros of such functions from $[0, 1]^2$ to $\mathbf{R}$ can be characterized precisely as strongly P-recognizable, closed sets. First, let us define the notion of the inverse modulus of continuity.

DEFINITION 5.1. *A function* $f : [0, 1]^2 \to \mathbf{R}$ *has a* polynomial inverse modulus at zeros *if there exist a polynomial* p *and a constant* n_0 *such that for all* $n \geq n_0$, *if* $|\mathbf{z} - \mathbf{z}_0| > 2^{-n}$ *for all zeros* $\mathbf{z}_0$ *of* f, *then* $|f(\mathbf{z})| > 2^{-p(n)}$. *The function* p *is called an* inverse modulus function at zeros.

The condition that f has a polynomial inverse modulus at zeros means that if we know that $f(\mathbf{z})$ is close to 0, then $\mathbf{z}$ must actually close to some zero $\mathbf{z}_0$. Thus, the conventional zero-testing of $|f(\mathbf{z})| \leq \epsilon$ works for the function f. Note that if f is not known to have a polynomial inverse modulus at zeros, then it may have zeros of arbitrarily high complexity (see [11]). Therefore, this condition is necessary if we are interested at zeros of reasonably low complexity.

THEOREM 5.2. *A closed set* $S \subseteq [0, 1]^2$ *is strongly P-recognizable iff there exists a P-computable function* $f : [0, 1]^2 \to \mathbf{R}$ *that has a polynomial inverse modulus at zeros such that* S *is exactly the set of zeros of* f.

Proof. First let $f : [0, 1]^2 \to \mathbf{R}$ be P-computable and have a polynomial inverse modulus q at zeros. Assume that M_1 is an oracle machine computing f. Let $S = \{\mathbf{z} : f(\mathbf{z}) = 0\}$. Consider the following oracle machine M for S.

> ORACLE MACHINE M. With oracles (ϕ, ψ) and input $n > n_0$, M simulates $M_1^{\phi,\psi}(q(n + 1))$. If M_1 outputs a number d such that $|d| \leq 2^{-q(n+1)}$ then M outputs 1, else M outputs 0.

Let $\mathbf{z}$ be the point in $\mathbf{R}^2$ represented by (ϕ, ψ). Assume that $\mathbf{z}$ is a zero of f; then the value d computed by $M_1^{\phi,\psi}(q(n + 1))$ must satisfy $|d| \leq 2^{-q(n+1)}$, and hence M must output 1.

On the other hand, if $\mathbf{z}$ has a distance greater than 2^{-n} from any zero of f, then $|f(\mathbf{z})| > 2^{-q(n)}$, and hence $|d| > 2^{-q(n)} - 2^{-q(n+1)} \geq 2^{-q(n+1)}$ and M must output 0. Thus, the errors of M can occur only within distance 2^{-n} of a zero $\mathbf{z}_0$. Since S is a closed set, this implies that the errors only occur within distance 2^{-n} of the boundary Γ_S.

Conversely, assume that $S \subseteq [0, 1]^2$ is a closed, strongly P-recognizable set. Assume that M is an oracle machine that strongly P-recognizes set S with a polynomial running time p. We are going to construct a function $f : [0, 1]^2 \to \mathbf{R}$ that satisfies the conditions of the theorem such that $S = \{\mathbf{z} : f(\mathbf{z}) = 0\}$. We assume that $p(n + 1) \geq p(n) + 2$.

We define function f via a sequence of discrete functions $\{\phi_n\}$, where each ϕ_n maps a point $\mathbf{d}$ in $A_n = (\mathbf{D}_{p(n)} \cap [0, 1])^2$ to a point $e \in \mathbf{D}$. The sequence $\{\phi_n\}$ will be defined recursively. First, $\phi_0(\mathbf{d}) = 0$ for all $\mathbf{d} \in A_0$. Next, we assume that ϕ_n has been defined on all $\mathbf{d} \in A_n$, and we extend it to a function ψ_n on $[0, 1]^2$ so that $\psi_n(\mathbf{z})$ is linearly defined from ϕ_n. In other words, assume that $\mathbf{z} = \langle x, y\rangle$ satisfies $a_1 \leq x < a_1 + 2^{-p(n)} = a_2$, $b_1 \leq y < b_1 + 2^{-p(n)} = b_2$ for some $a_1, b_1 \in \mathbf{D}_{p(n)} \cap [0, 1]$. That is, $\langle a_1, b_1\rangle$, $\langle a_1, b_2\rangle$, $\langle a_2, b_1\rangle$, $\langle a_2, b_2\rangle$ form a square of size $2^{-p(n)} \times 2^{-p(n)}$ that contains $\mathbf{z}$. Then,

$$\begin{aligned}\psi_n(\mathbf{z}) = \; & (1 - r) \cdot (1 - s) \cdot \phi_n(\langle a_1, b_1\rangle) + (1 - r) \cdot s \cdot \phi_n(\langle a_1, b_2\rangle) \\ & + r \cdot (1 - s) \cdot \phi_n(\langle a_2, b_1\rangle) + r \cdot s \cdot \phi_n(\langle a_2, b_2\rangle),\end{aligned}$$

where $r = (x - a_1) \cdot 2^{p(n)}$ and $s = (y - b_1) \cdot 2^{p(n)}$.

Now, we define $\phi_{n+1}(\mathbf{d})$ for all $\mathbf{d} \in A_{n+1}$ in terms of function ψ_n. For each point $\mathbf{d} \in A_{n+1}$, exactly one of the following cases may occur:

(i) $\mathbf{d} \in A_n$;

(ii) there exist two points $\mathbf{e}_1, \mathbf{e}_2 \in A_n$ that lie in a horizontal or a vertical line with $|\mathbf{e}_1 - \mathbf{e}_2| = 2^{-p(n)}$, and $\mathbf{d}$ lies strictly between $\mathbf{e}_1$ and $\mathbf{e}_2$;

(iii) there exist $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \mathbf{e}_4$ in A_n that are the four corners of a square of size $2^{-p(n)} \times 2^{-p(n)}$ such that $\mathbf{d}$ is an interior point of this square.

In each case, we call these points in A_n the *parent points* of $\mathbf{d}$. The value $\phi_{n+1}(\mathbf{d})$ is defined from the values of ψ_n on its parents as follows:

(1) If all parents $\mathbf{e}$ of $\mathbf{d}$ have $\phi_n(\mathbf{e}) > 0$, then $\phi_{n+1}(\mathbf{d}) = \psi_n(\mathbf{d})$.

(2) Otherwise, evaluate $M^{\mathbf{d}}(n)$, and let

$$\phi_{n+1}(\mathbf{d}) = \begin{cases} \psi_n(\mathbf{d}) & \text{if } M^{\mathbf{d}}(n) = 1, \\ 2^{-n} & \text{if } M^{\mathbf{d}}(n) = 0. \end{cases}$$

We observe the following properties of $\{\phi_n\}$.

(a) If $\mathbf{d} \in A_n$ and $\delta(d, S) < 2^{-p(n)}$, then $\phi_n(d) = 0$.

Proof. We prove this by induction on n. First, $\phi_0(\mathbf{d}) = 0$ for all $\mathbf{d} \in A_0$, so the claim holds for $n = 0$. Next, for $n > 0$, observe that if $\delta(\mathbf{d}, S) \leq 2^{-p(n)}$, then all parent points $\mathbf{e}$ of $\mathbf{d}$ have $\delta(\mathbf{e}, S) \leq 2^{-(p(n-1))}$. So, by the inductive hypothesis, we must have $\phi_{n-1}(\mathbf{e}) = 0$, and the value of $\phi_n(\mathbf{d})$ is determined by case (2). Note that $\delta(\mathbf{d}, S) \leq 2^{-p(n)}$, so there exists a point $\mathbf{z} \in S$ that is also in $N(\mathbf{d}; 2^{-p(n)})$. From Lemma 3.4 (we noted that it also holds for P-recognizable sets), we know that if $M^{\mathbf{d}}(n) = 0$, then $N(\mathbf{d}; 2^{-p(n)}) \subseteq S^c \cup E_n(M)$, and hence $\mathbf{z} \in E_n(M)$. However, from the assumption that M strongly recognizes S, $\mathbf{z}$ cannot be in $E_n(M)$. Therefore, we conclude that $M^{\mathbf{d}}(n)$ must be equal to 1, and hence $\phi_n(\mathbf{d}) = \psi_{n-1}(\mathbf{d}) = 0$. □

(b) If $\mathbf{d} \in A_n$ and $\delta(\mathbf{d}, S) > 2^{-n}$, then $\phi_n(\mathbf{d}) > 0$.

Proof. If all the parent points $\mathbf{e}$ of $\mathbf{d}$ have values $\phi_{n-1}(\mathbf{e}) > 0$, then $\phi_n(\mathbf{d}) > 0$. Otherwise, $\phi_n(\mathbf{d})$ is determined by case (2). Since $\delta(\mathbf{d}, S) > 2^{-n}$, $M^{\mathbf{d}}(n)$ must output 0, and hence $\phi_n(\mathbf{d}) = 2^{-n}$. □

(c) If $\mathbf{d}_1, \mathbf{d}_2 \in A_n$ and $|\mathbf{d}_1 - \mathbf{d}_2| = 2^{-p(n)}$, then $|\phi_n(\mathbf{d}_1) - \phi_n(\mathbf{d}_2)| \leq 2^{-n}$.

Proof. We consider the following cases.

Case 1. $\phi_n(\mathbf{d}_i) = \psi_{n-1}(\mathbf{d}_i)$ for both $i = 1, 2$. Note that the parents of $\mathbf{d}_1$ and $\mathbf{d}_2$ must be the corners of a square of size $2^{-p(n-1)} \times 2^{-p(n-1)}$. Then, by the inductive hypothesis, the values of these corners do not differ more than $2^{-(n-1)}$. By the linearity of ψ_{n-1}, we see that the values $\phi_n(\mathbf{d}_1)$ and $\phi_n(\mathbf{d}_2)$ do not differ more than $1/2 \cdot 2^{-(n-1)} = 2^{-n}$, since $p(n) \geq p(n-1) + 2$.

Case 2. $\phi_n(\mathbf{d}_1) = \phi_n(\mathbf{d}_2) = 2^{-n}$. Then, it is evident that (c) holds.

Case 3. $\phi_n(\mathbf{d}_1) = \psi_{n-1}(\mathbf{d}_1)$ and $\phi_n(\mathbf{d}_2) = 2^{-n} \neq \psi_{n-1}(\mathbf{d}_2)$. This can happen only when one of the parent points $\mathbf{e}$ of $\mathbf{d}_2$ has $\phi_{n-1}(\mathbf{e}) = 0$. By the inductive hypothesis, all the parents $\mathbf{e}'$ of both $\mathbf{d}_1$ and $\mathbf{d}_2$ have values $\phi_{n-1}(\mathbf{e}') \leq 2^{-(n-1)}$, since $|\mathbf{e}' - \mathbf{e}| \leq 2^{-p(n-1)}$. Thus $\phi_n(\mathbf{d}_1) \leq 2^{-(n-1)}$, so $|\phi_n(\mathbf{d}_1) - \phi_n(\mathbf{d}_2)| \leq 2^{-n}$. □

(d) For any $\mathbf{d} \in A_n$, $|\psi_{n-1}(\mathbf{d}) - \phi_n(\mathbf{d})| \leq 2^{-n}$.

Proof. As we proved above in Case 3 of (c), if $\phi_n(\mathbf{d}) \neq \psi_{n-1}(\mathbf{d})$, then it must be the case that $0 \leq \psi_{n-1}(\mathbf{d}) \leq 2^{-(n-1)}$ and $\phi_n(\mathbf{d}) = 2^{-n}$. □

Now, we define $f(\mathbf{z}) = \lim_{n \to \infty} \psi_n(\mathbf{z})$. By properties (c) and (d) above, we can see that f is well defined, continuous, and has a polynomial modulus function p. By properties (a) and (b), we see that $f(\mathbf{z}) = 0$ iff $\mathbf{z} \in S$.

Finally, we observe the following lower bound for $\phi_n(\mathbf{d})$.

(e) If $\mathbf{d} \in A_n$ and $\phi_n(\mathbf{d}) > 0$, then $\phi_n(\mathbf{d}) \geq 2^{-(n+q(n))}$, where $q(n) = \sum_{i=1}^{n} 2p(i)$.

Proof. If $\phi_n(\mathbf{d})$ is defined as 2^{-n}, then (e) holds. Otherwise, $\mathbf{d}$ must be equal to $\psi_{n-1}(\mathbf{d})$, which is linearly defined from the values $\phi_{n-1}(\mathbf{e})$ of the ancestor points $\mathbf{e}$ of $\mathbf{d}$. Therefore, at least one of the parent points $\mathbf{e}$ has $\phi_{n-1}(\mathbf{e}) > 0$ and, by the inductive hypothesis, it is at least $2^{-(n-1+q(n-1))}$. Since $\phi_n(\mathbf{d})$ is defined linearly from the parent values, its value is at least $2^{-2p(n)}$ times the value of $\phi_{n-1}(\mathbf{e})$. The bound $2^{-(n+q(n))}$ follows. □

Now we can see that f has a polynomial inverse modulus at zeros. If $\delta(\mathbf{z}, S) > 2^{-n}$, then for all points $\mathbf{d} \in A_{n+1}$ such that $|\mathbf{d} - \mathbf{z}| \leq 2^{-p(n+1)}$, we have $\delta(\mathbf{d}, S) > 2^{-(n+1)}$, and hence $M^{\mathbf{d}}(n+1) = 0$. This implies that $\phi_{n+1}(\mathbf{d}) > 0$. By property (e), $\phi_{n+1}(\mathbf{d}) \geq 2^{-(n+q(n))}$ for all these points $\mathbf{d}$ and, by the definition of $\{\phi_n\}$, $\phi_{n+k}(\mathbf{d}) = \phi_{n+1}(\mathbf{d}) \geq 2^{-(n+q(n))}$ for all $k \geq 1$. It follows that $\psi_{n+k}(\mathbf{z}) \geq 2^{-(n+q(n))}$ for all $k \geq 1$, so $f(\mathbf{z}) \geq 2^{-(n+q(n))}$. □

The above characterization, together with Theorem 4.6, also gives an upper bound to the complexity of isolated zeros of functions having polynomial inverse moduli at zeros; namely, such isolated zeros must be *NP*-computable. To get a better understanding of the complexity of such isolated zeros, we summarize the known results on the relationship between *NP*-computable real numbers and P-computable real numbers in the following.

Recall that a tally set is a set over a singleton alphabet $\{0\}$. Also, recall that *UP* is the class of sets accepted by unambiguous nondeterministic machines in polynomial time; that is, for any input w, there is at most one accepting computation for w. The relation between the class *UP* and the class *P* is related to the existence of one-way functions. We say a function $\phi : \{0, 1\}^* \to \{0, 1\}^*$ is a *one-way function* if ϕ is one-to-one, P-computable, polynomially-honest,[3] and there is no P-computable function ψ satisfying $\psi(\phi(w)) = w$ for all $w \in \{0, 1\}^*$. It is known that $P = UP$ is equivalent to the the condition that one-way functions do not exist. In the following, we will use the condition that all tally sets in the complexity class $UP \cap coUP$ are actually in P. This condition is equivalent to the condition that there do not exist stronger one-way functions $\phi : \{0, 1\}^* \to \{0, 1\}^*$ having the following property: $\{0\}^* \subseteq \text{Range}(\phi)$ and ϕ^{-1} is not P-computable on $\{0\}^*$.

COROLLARY 5.3. *In the following,* (a) $\Rightarrow$ (b) $\iff$ (c) $\Rightarrow$ (d).

(a) *All tally sets in* Δ_2^P *are in P.*

(b) *All NP-computable real numbers are P-computable.*

(c) *If* $f : [0, 1]^2 \to \mathbf{R}$ *is P-computable and has a polynomial inverse modulus at zeros, then all isolated zeros of f are P-computable.*

(d) *All tally sets in* $UP \cap coUP$ *are in P.*

In the above, the direction (a) $\Rightarrow$ (b) is quite easy to see. The direction (c) $\Rightarrow$ (d) requires a more involved construction, and was proved in [8] and [9]. Whether the above complexity gap between Δ_2^P and $UP \cap coUP$ can be narrowed is an open question.

6. Winding numbers. In the previous sections, we introduced some notions of polynomial time computable subsets of $\mathbf{R}^2$. For bounded, simply connected regions S in $\mathbf{R}^2$, there is another natural representation for it, that is, the boundary Γ_S of the region. In the next section, we are going to study the membership problem corresponding to the boundary representation. That is, if the boundary Γ_S of a region S is given as a polynomial-time computable curve, what is the complexity of the membership problem for S? We approach this problem by first studying a more general problem of counting the winding numbers. The winding number problem is, informally, the problem of computing the winding number of a polynomial-time computable closed curve with respect to a given point that is not on the curve. For a simple, closed curve, the winding number determines whether a point is in the interior or the exterior of the curve. Thus, the upper bound for the winding number problem is also an upper bound for the membership problem with respect to the boundary representation.

The notion of the winding number can be formally defined as follows: Let $\arg(\mathbf{z})$ denote the arguments of $\mathbf{z} \in \mathbf{R}^2$ if $\mathbf{z} \neq \langle 0, 0\rangle$; that is, arg is a multivalued function from $\mathbf{R}^2 - \{\langle 0, 0\rangle\}$ to $\mathbf{R}$ such that if $\mathbf{z} = \langle x, y\rangle$ then $x = |\mathbf{z}| \cos(\arg(\mathbf{z}))$ and $y = |\mathbf{z}| \sin(\arg(\mathbf{z}))$. Let Γ be a closed curve with a representation f; that is, f is a continuous function from $[0, 1]$ to $\mathbf{R}^2$ such that $f(0) = f(1)$, and Γ is the range of f. For any point $\mathbf{z}_0 \notin \Gamma$, a *continuous argument function* $h_{\mathbf{z}_0}$ is a continuous function such that $h_{\mathbf{z}_0}(t)$ is a value of $\arg(f(t) - \mathbf{z}_0)$. It is easy to see that any two continuous argument functions differ by a multiple of 2π. The winding number of Γ with respect to $\mathbf{z}_0$ is defined as

$$\text{wind}_\Gamma(\mathbf{z}_0) = \frac{1}{2\pi}(h_{\mathbf{z}_0}(1) - h_{\mathbf{z}_0}(0))$$

[3]That is, there is a polynomial q such that $q(\ell(\phi(w))) \geq \ell(w)$ for all $w \in \{0, 1\}^*$.

for *any* continuous argument function $h_{\mathbf{z}_0}$. Equivalently, we may also define the winding number in the form of integrals over the curve Γ:

$$\mathrm{wind}_\Gamma(\mathbf{z}_0) = \frac{1}{2\pi i}\int_\Gamma \frac{1}{\mathbf{z}-\mathbf{z}_0}\,d\mathbf{z}.$$

Each of these definitions provides a natural method for computing the winding number. Our algorithm in Theorem 6.4 is based on the first method presented in Henrici [5].

Note that the winding number $\mathrm{wind}_\Gamma(\mathbf{z})$ of a curve Γ with respect to a point $\mathbf{z} \in \mathbf{R}^2$ regarded as a function of $\mathbf{z}$ has discontinuities on the curve Γ. Thus, as in polynomial-time computable subsets of $\mathbf{R}^2$, any notion of computability for winding numbers must allow errors to occur. Our computational model for winding numbers is similar to the model for P-recognizable sets: it allows the errors but only when the input point is close to the curve, because the discontinuity of the winding number function occurs exactly around the curve Γ. More formally, let $f : [0, 1] \to \mathbf{R}^2$ be a polynomial-time computable function that represents a closed curve Γ; that is, Γ is the image of function f on $[0, 1]$. We say an oracle Turing machine M *computes the winding number* of Γ if, for all oracles (ϕ, ψ) that represent some $\mathbf{z}$ in $\mathbf{R}^2$, and for all inputs n, $M^{\phi,\psi}(n)$ outputs the winding number of $\mathbf{z}$ with respect to Γ whenever $\delta(\mathbf{z}, \Gamma) > 2^{-n}$. We say the winding number of a closed curve Γ is *polynomial-time computable* if there exists such an oracle machine that operates in polynomial time.

Note that, if $\delta(\mathbf{z}_1, \Gamma) > 2^{-n}$ and $|\mathbf{z}_2 - \mathbf{z}_1| \leq 2^{-(n+1)}$, then the winding numbers of $\mathbf{z}_1$ and $\mathbf{z}_2$ are equal. Thus, as far as the polynomial-time computability of the winding number is concerned, we only need to make sure that the machine M works for all dyadic points.

PROPOSITION 6.1. *The winding number of a closed curve Γ is polynomial-time computable iff there exists a Turing machine such that $M(\mathbf{d}, n) = \mathrm{wind}_\Gamma(\mathbf{d})$ for all $\mathbf{d} \in \mathbf{D} \times \mathbf{D}$ and all $n \in \mathbf{N}$ satisfying $\delta(\mathbf{d}, \Gamma) > 2^{-n}$.*

The complexity of the winding number problem will be characterized by the counting class $\#P$. The following properties of the class $\#P$ are well known (see, for instance, [9]).

PROPOSITION 6.2. (a) *For any P-computable function $F : \{0, 1\}^* \times \{0, 1\}^* \to \mathbf{N}$, the function $G : \{0, 1\}^* \times \mathbf{N} \to \mathbf{N}$, defined by $G(w_1, n) = \sum_{|w_2|=n} F(w_1, w_2)$, is in $\#P$.*

(b) *For any function $G : \{0, 1\}^* \to \mathbf{N}$ that is in $\#P$, there exist a set $A \subseteq \{0, 1\}^*$ in P and a polynomial p such that for each $w \in \{0, 1\}^*$, $G(w)$ is equal to the number of strings u of length $p(\ell(w))$ such that $\langle u, w\rangle \in A$.*

Let Γ be a closed curve represented by a continuous function $f : [0, 1] \to \mathbf{R}^2$. Recall that the winding number $\mathrm{wind}_\Gamma(\mathbf{z}_0)$ of Γ with respect to a point $\mathbf{z}_0 \notin \Gamma$ is equal to $(h_{\mathbf{z}_0}(1) - h_{\mathbf{z}_0}(0))/2\pi$, where $h_{\mathbf{z}_0}$ is any continuous argument function with respect to $\mathbf{z}_0$. We first show that the *local* argument increase of the function $h_{\mathbf{z}_0}$ is computable in polynomial time if f is P-computable. Let $g(\mathbf{z}_0, t_1, t_2)$ denote the value $h_{\mathbf{z}_0}(t_2) - h_{\mathbf{z}_0}(t_1)$. Note that the function g is independent of the choice of the particular continuous argument function $h_{\mathbf{z}_0}$.

LEMMA 6.3. *Let $f : [\alpha, \beta] \to \mathbf{R}^2$ be a P-computable function that represents an arc Γ, and $\mathbf{d}$ be a dyadic point of $\mathbf{R}^2$ not in Γ. Assume that there exists an $\epsilon > 0$ such that $\delta(\mathbf{d}, \Gamma) > \epsilon$, and for all $t_1, t_2 \in [\alpha, \beta]$, $|f(t_1) - f(t_2))| \leq \epsilon$. Then, the argument increase function $g(\mathbf{d}, \alpha, \beta)$ is P-computable.*

Proof. Since $\delta(\mathbf{d}, \Gamma) > \epsilon$, Γ lies outside the square $N(\mathbf{d}; \epsilon)$. Furthermore, $|f(t_1) - f(t_2)| \leq \epsilon$ for all $t_1, t_2 \in [\alpha, \beta]$. This implies that a continuous argument function $h_{\mathbf{d}}$ must have difference $|h_{\mathbf{d}}(\alpha) - h_{\mathbf{d}}(\beta)| \leq \pi/2$. Thus, we only need to obtain any values $a \in \arg(f(\alpha) - \mathbf{d})$ and $b \in \arg(f(\beta) - \mathbf{d})$. Then, let their difference be γ, and γ' be the number in $[-\pi/4, \pi/4]$ such that $\gamma' \equiv \gamma \pmod{2\pi}$. Output γ'.[4] □

[4]More precisely, we should take care of the rounding errors when we evaluate a and b and compute γ' from γ. These are routine works, and we leave them to the reader.

THEOREM 6.4. *For any continuous closed curve Γ that has a polynomial-time representation f, there exists an oracle machine that computes the winding number of Γ in polynomial time using a function G in $\#P$ as the oracle. Therefore, the winding number of a P-computable curve Γ is always P-computable if $FP = \#P$.*

Proof. By Proposition 6.1, we only need to construct an oracle machine that, on inputs $(\mathbf{d}, n)$ with $\mathbf{d} \in \mathbf{D} \times \mathbf{D}$ and n an integer and an oracle $G \in \#P$, computes $\text{wind}_\Gamma(\mathbf{d})$ whenever $\delta(\mathbf{d}, \Gamma) > 2^{-n}$.

Assume that f has a polynomial modulus function p. Then, for any two numbers $t_1, t_2 \in [0, 1]$ such that $|t_1 - t_2| \leq 2^{-p(n)}$, we must have $|f(t_1) - f(t_2)| \leq 2^{-n}$. From Lemma 6.3, there exists a polynomial-time machine M that computes the argument increase $g(\mathbf{d}, \alpha, \beta)$ whenever $0 \leq \beta - \alpha \leq 2^{-p(n)}$, since $\delta(\mathbf{d}, \Gamma) > 2^{-n}$. We define a P-computable function $F : (\mathbf{D} \cap [0, 1]) \times \{0, 1\}^* \to \mathbf{N}$ as follows.

> ALGORITHM FOR FUNCTION F. On input $\mathbf{d}$ and w, $\ell(w) = p(n)$,[5] let i_w be the integer whose n-bit binary representation is w. Let $t_1 = i_w \cdot 2^{-p(n)}$ and $t_2 = t_1 + 2^{-p(n)}$. Simulate M on input $(\mathbf{d}, t_1, t_2)$ with the error bound $2^{-2p(n)}$ to get a dyadic rational e such that $|e - g(\mathbf{d}, t_1, t_2)| \leq 2^{-2p(n)}$. (Note that $-\pi/2 < e < \pi/2$.) Next, compute a dyadic rational e' of length $2^{2p(n)}$ such that $|e' - (e/2\pi + 1)| \leq 2^{-2p(n)}$, and let $F(\mathbf{d}, w) = e' \cdot 2^{2p(n)}$.

It is clear that

$$a = \left(\sum_{\ell(w)=p(n)} F(\mathbf{d}, w) \right) \cdot 2^{-2p(n)} - 2^{p(n)}$$

is close to $\text{wind}_\Gamma(\mathbf{d})$ within an error $2^{-(p(n)-1)}$. Therefore, by Proposition 6.2(a) we can ask the oracle for the value $G(\mathbf{d}, p(n)) = \sum_{\ell(w)=p(n)} F(\mathbf{d}, w)$ and find the closest integer to a, which is the winding number of $\mathbf{d}$. □

Conversely, we can show that the $\#P$-complete oracle G is necessary for computing the winding number in polynomial time.

THEOREM 6.5. *For any function $G \in \#P$, there exist a P-computable function $f : [0, 1] \to \mathbf{R}^2$ that computes a closed curve Γ, a P-computable (discrete) function $\phi : \{0, 1\}^* \to \mathbf{D} \times \mathbf{D}$, and a polynomial p such that*

(i) $\delta(\phi(w), \Gamma) \geq 2^{-p(\ell(w))}$ *for all* $w \in \{0, 1\}^*$, *and*

(ii) *the winding number of the curve Γ with respect to the point $\phi(w)$ is equal to $G(w)$.*

Proof. We first describe a basic construction that will be used later. For any integer n and any set $B \subseteq \{0, 1\}^n$, we define an arc Γ_B that is represented by a function $g_B : [0, 1] \to \mathbf{R}^2$. For each integer k, $0 \leq k \leq 2^n - 1$, we let u_k denote the n-bit binary representation of k, and $t_k = 1/4 + k \cdot 2^{-(n+1)}$.

(1) g_B is linear on $[0, 1/4]$: $g_B(0) = \langle -2, 0 \rangle$ and $g_B(1/4) = \langle -1 - 2 \cdot 2^{-n}, 0 \rangle$.

(2) For each k such that $0 \leq k \leq 2^n - 1$, if $u_k \notin B$, then g_B is linear on $[t_k, t_{k+1}]$: $g_B(t_k) = \langle -1 + (k-2) \cdot 2^{-n}, 0 \rangle$ and $g_B(t_{k+1}) = \langle -1 + (k-1) \cdot 2^{-n}, 0 \rangle$.

(3) For each k such that $0 \leq k \leq 2^n - 1$, if $u_k \in B$, then g_B is piecewise linear on $[t_k, t_{k+1}]$: it divides $[t_k, t_{k+1}]$ into five subintervals of equal length and maps them to the five consecutive line segments defined by the following points:

$$\begin{array}{ll} \langle -1 + (k-2) \cdot 2^{-n}, 0 \rangle, & \langle -1 + (k-2) \cdot 2^{-n}, 1 - (k-2) \cdot 2^{-n} \rangle, \\ \langle 1 - (k-2) \cdot 2^{-n}, 1 - (k-2) \cdot 2^{-n} \rangle, & \langle 1 - (k-2) \cdot 2^{-n}, -1 + (k-2) \cdot 2^{-n} \rangle, \\ \langle -1 + (k-1) \cdot 2^{-n}, -1 + (k-2) \cdot 2^{-n} \rangle, & \langle -1 + (k-1) \cdot 2^{-n}, 0 \rangle. \end{array}$$

(4) g_B is linear on $[3/4, 1]$: $g_B(3/4) = \langle -2 \cdot 2^{-n}, 0 \rangle$ and $g_B(1) = \langle 2, 0 \rangle$.

[5] For simplicity, we only define F on w of length $p(n)$ for some n.

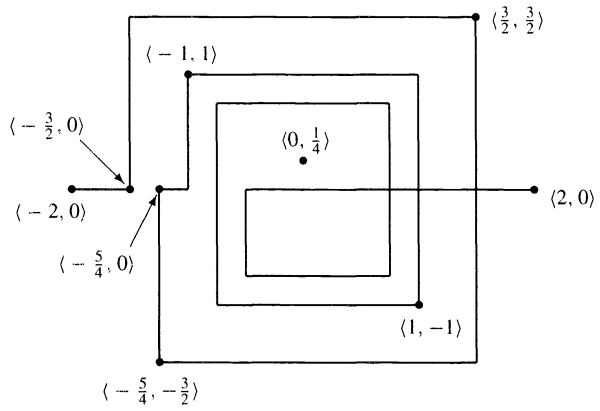


FIG. 1. *The arc Γ_B for set $B = \{00, 10, 11\}$.*

In Fig. 1, we show the arc g_B for the set $B = \{00, 10, 11\}$, where $n = 2$.

It is easy to see that if we connect the two endpoints of the arc g_B from below to form a closed curve (called the *extended curve* g_B), then the winding number of the extended curve with respect to the point $\langle 0, 2^{-n}\rangle$ is equal to the size of B.

Now we describe the function f. For convenience, we will define f on $[0, 2]$ instead of $[0, 1]$. It is clear we can easily transform it to a function on $[0, 1]$ if necessary. Since $G \in \#P$, there exist by Proposition 6.2(b) a set $A \in P$ and a polynomial q such that for all $w \in \{0, 1\}^*$ with $\ell(w) = n$, and $G(w)$ is equal to the size of set $B_w = \{u : \ell(u) = q(\ell(w)), \langle u, w\rangle \in A\}$. Recall that in the proof of Theorem 3.9 we defined for each $w \in \{0, 1\}^*$ a real number x_w with the property that $x_w + 2^{-2n} = x_u$, where $n = \ell(w)$ and u is the lexicographic successor of w.

For each $w \in \{0, 1\}^*$ of length n, the function f on the subinterval $[x_w, x_w + 2^{-2n}]$ is a linear transformation of g_{B_w} on $[0, 1]$. Let $g_1, g_2 : [0, 1] \to \mathbf{R}$ be such that $g_{B_w}(t) = \langle g_1(t), g_2(t)\rangle$. Then, f on $[x_w, x_w + 2^{-2n}]$ can be defined as follows:

$$f(t) = \langle 2^{-(2n+2)} \cdot g_1(2^{2n}(t - x_w)) + x_w + 2^{-(2n+1)}, 2^{-(2n+2)} \cdot g_2(2^{2n}(t - x_w))\rangle.$$

f is now defined on $[0, 1]$. Now, we define f on $[1, 2]$ as piecewise linear mapping the interval $[1, 2]$ to three line segments defined by the following four points: $\langle 1, 0\rangle$, $\langle 1, -1\rangle$, $\langle 0, -1\rangle$, and $\langle 0, 0\rangle$.

It is clear that f is continuous on $[0, 1]$ and defines a closed curve Γ. It is also easy to see that if we define $\phi(w) = \langle x_w + 2^{-(2n+1)}, 2^{-(2n+q(n)+2)}\rangle$, where $n = \ell(w)$, then ϕ is P-computable and $\delta(\phi(w), \Gamma) = 2^{-(2n+q(n)+2)}$. Furthermore, the winding number of Γ with respect to $\phi(w)$ is equal to that of the extended curve defined by g_{B_w} with respect to the point $\langle 0, 2^{-q(n)}\rangle$, which is equal to $G(w)$.

Finally, we show that f is P-computable. First, it is clear that f is P-computable on $[1, 2]$, since it maps $[1, 2]$ to three line segments piecewise linearly. Next, we show that f on $\mathbf{D} \cap [0, 1]$ is P-computable as a discrete function. To see this, we check that the functions g_{B_w} are uniformly computable in the sense that, for any $w \in \{0, 1\}^*$ and $t \in \mathbf{D} \cap [0, 1]$, we can compute $g_{B_w}(t)$ in polynomial time: We first decide whether $t \in [0, 1/4]$, $t \in [3/4, 1]$, or $t \in [t_k, t_{k+1}]$ for some k, $0 \le k \le 2^n - 1$. In the first two cases, the computation is then straightforward and, for the last case, the computation is also easy as long as we know whether $u_k \in B_w$. Since f is a simple linear transformation of g_{B_w}, the uniform P-computability of g_{B_w} on $\mathbf{D} \cap [0, 1]$ implies that f is also P-computable on $\mathbf{D} \cap [0, 1]$.

By Proposition 2.2, we are left to show that f has a polynomial modulus of continuity. First, we note that for any set $B \subseteq \{0,1\}^n$, the function g_B has the property that, for all $k > 0$, if $t_1, t_2 \in [0,1]$ and $|t_1 - t_2| \leq 2^{-k}$, then $|g_B(t_1) - g_B(t_2)| \leq 2^{-(k-n-6)}$ since g_B is piecewise linear and maps subintervals of length $\geq 2^{-(n+1)}/5$ to line segments of length ≤ 4. Following this observation, we claim that if $t_1, t_2 \in [0,1]$ and $0 < t_2 - t_1 \leq 2^{-(3k+q(k))}$, then $|f(t_1) - f(t_2)| \leq 2^{-(k-5)}$. (Recall that $a_k = 1 - 2^{-(k-1)}$.)

Case 1. $t_1, t_2 \in [a_k, 1]$. Then we must have $|f(t_i) - \langle 1, 0\rangle| \leq 2^{-(k-1)}$ for both $i = 1, 2$. Thus $|f(t_1) - f(t_2)| \leq 2^{-(k-2)}$.

Case 2. $t_1, t_2 \in [x_w, x_w + 2^{-2n}]$ for some w of length $n \leq k$. Then $|t_1' - t_2'| \leq 2^{-(k+q(k))}$, where $t_i' = 2^{2n}(t_i - x_w)$ for $i = 1, 2$, and so from the above observation on g_{B_w}, we know that $|g_{B_w}(t_1') - g_{B_w}(t_2')| \leq 2^{-(k-6)}$. It follows that $|f(t_1) - f(t_2)| \leq 2^{-(2n+k-4)}$.

Case 3. $t_1 < x_w \leq t_2$ for some w of length $n \leq k$. Then t_1 must be in $[x_u, x_u + 2^{-2\ell(u)}]$, where u is the lexicographic predecessor of w and $x_u + 2^{-2\ell(u)} = x_w$. Then, applying Case 2 to t_1 and x_w, and x_w and t_2, we get

$$|f(t_1) - f(t_2)| \leq |f(t_1) - f(x_w)| + |f(t_2) - f(x_w)| \leq 2^{-(k-5)}.$$

This completes the proof of the claim and hence the proof of the theorem. □

COROLLARY 6.6. *The following are equivalent:*

(a) $FP = \#P$.

(b) *For every P-computable closed curve* Γ, *the winding number problem of* Γ *is solvable in polynomial time.*

7. The membership problem. Now we go back to the membership problem for a simply connected region S represented by a P-computable boundary Γ_S. It is well known that the winding number of the simple closed curve Γ_S with respect to a point $\mathbf{z} \notin \Gamma_S$ is 1 or -1 if $\mathbf{z}$ is inside the set S and 0 if $\mathbf{z}$ is not in $S \cup \Gamma_S$. Thus, the upper bound $\#P$ for the winding number problem is also an upper bound for this membership problem. In fact, we can apply the proof of Theorem 6.4 to give a slightly tighter upper bound. That is, we need only one bit from a function in $\#P$ to help us determine the membership of a given point. This observation is closely related to the recently studied complexity class *MidBitP* [17], [4].

COROLLARY 7.1. *Let* $f : [0,1] \to [0,1]^2$ *be a P-computable function defining a simple closed curve* Γ. *Then, the interior S of the curve is P-recognizable with respect to an oracle* $G \in \#P$. *In addition, the oracle machine M that P-recognizes S needs only to ask the oracle for one bit of a value of G.*

Proof. In the proof of Theorem 6.4, modify the function F to $F'(\mathbf{d}, w) = F(\mathbf{d}, w) + 4$. Note that

$$\sum_{\ell(w)=p(n)} F(\mathbf{d}, w) \cdot 2^{-2p(n)} = 2^{p(n)} + \mathrm{wind}_\Gamma(\mathbf{d}) + \epsilon$$

for some ϵ of size $|\epsilon| \leq 2^{-(p(n)-1)}$. So

$$\sum_{\ell(w)=p(n)} F'(\mathbf{d}, w) \cdot 2^{-2p(n)} = 2^{p(n)} + \mathrm{wind}_\Gamma(\mathbf{d}) + \epsilon'$$

for some ϵ' such that $0 < \epsilon' < 2^{-(p(n)-3)}$. That is, the integral part of $\sum F'(\mathbf{d}, w) \cdot 2^{-2p(n)}$ is $2^{p(n)} + 1$ or $2^{p(n)} - 1$ if $\mathbf{d} \in S$, and $2^{p(n)}$ if $\mathbf{d} \notin S$. Or, equivalently, the $(2p(n)+1)$st (least significant) bit of $G'(\mathbf{d}, p(n)) = \sum_{\ell(w)=p(n)} F'(\mathbf{d}, w)$ is 1 iff $\mathbf{d} \in S$. □

For the lower bound of the membership problem, we can only prove a weaker bound of *UP*. The following is a simple characterization of the complexity class *UP*. (Also see the discussion on the relation between *P* and *UP* at the end of §5.)

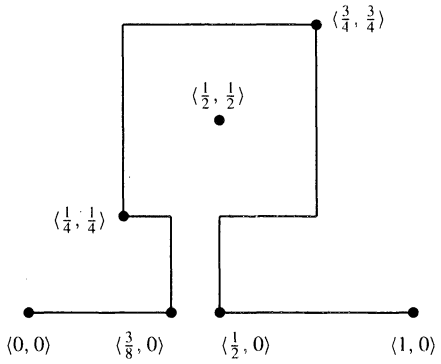


FIG. 2. *The arc Γ_B for set $B = \{01\}$.*

PROPOSITION 7.2. *A set $A \subseteq \{0, 1\}^*$ is in UP iff there exist a set $B \in P$ and a polynomial p such that for all $w \in \{0, 1\}^*$ of length n,*

$$w \in A \iff (\exists u, \ell(u) = p(\ell(w)))\ \langle w, u\rangle \in B$$
$$\iff (\exists \text{ unique } u, \ell(u) = p(\ell(w)))\ \langle w, u\rangle \in B.$$

THEOREM 7.3. *For any set $A \in UP$, there exist a P-computable function $f : [0, 1] \to \mathbf{R}^2$ that defines a simple closed curve Γ, a P-computable function $\phi : \{0, 1\}^* \to \mathbf{D}$, and some polynomial p such that for all $w \in \{0, 1\}^*$,*

(i) $\delta(\phi(w), \Gamma) \geq 2^{-p(\ell(w))}$ *for some fixed polynomial p, and*

(ii) $w \in A$ *iff* $\phi(w) \in S$.

Proof. The general idea of the construction is similar to that of Theorem 6.5. First, we describe a basic function $g_B : [0, 1] \to \mathbf{R}^2$ for each $B \subseteq \{0, 1\}^n$ that is either a singleton or empty. Again, for each $k \leq 2^n - 1$, we let u_k be the n-bit binary representation for k, and $t_k = 1/4 + k \cdot 2^{-(n+1)}$.

(1) g_B linearly maps $[0, 1/4]$ to the line segment from $\langle 0, 0\rangle$ to $\langle 1/4, 0\rangle$.

(2) For each k such that $0 \leq k \leq 2^n - 1$, if $u_k \notin B$, then g_B linearly maps $[t_k, t_{k+1}]$ to the line segment from $\langle t_k, 0\rangle$ to $\langle t_{k+1}, 0\rangle$.

(3) If $u_k \in B$, then g_B is piecewise linear on $[t_k, t_{k+1}]$: it divides $[t_k, t_{k+1}]$ into seven subintervals of equal length and maps them to seven consecutive line segments determined by the following breakpoints:

$$\langle t_k, 0\rangle, \quad \langle t_k, 1/4\rangle, \quad \langle 1/4, 1/4\rangle, \quad \langle 1/4, 3/4\rangle,$$
$$\langle 3/4, 3/4\rangle, \quad \langle 3/4, 1/4\rangle, \quad \langle t_{k+1}, 1/4\rangle, \quad \langle t_{k+1}, 0\rangle.$$

(4) g_B linearly maps $[3/4, 1]$ to the line segment from $\langle 3/4, 0\rangle$ to $\langle 1, 0\rangle$.

Figure 2 shows the function g_B with respect to set $B = \{01\}$, $n = 2$.

As in the functions g_B defined in Theorem 6.5, if we connect the points from $\langle 1, 0\rangle$ to $\langle 0, 0\rangle$ by an arc below these two points, then the point $\langle 1/2, 1/2\rangle$ is in the interior of the curve if B is a singleton and is not in the interior if B is empty.

Now we define the function f. First, for any set $B \subseteq \{0, 1\}^*$ and any string $w \in \{0, 1\}^*$, let $B_w = \{u : \ell(u) = p(\ell(w)), \langle w, u\rangle \in B\}$. Then, from Proposition 7.2, for set $A \in UP$ there exists a set $B \in P$ such that for each $w \in \{0, 1\}^*$, if $w \in A$ then B_w is a singleton set, and if $w \notin A$ then B_w is empty.

We define, for each w, the number x_w as in the proof of Theorem 3.9 such that $x_w + 2^{-2n} = x_u$, where u is the lexicographic successor of w. We define the function f on each interval $[x_w, x_w + 2^{-2n}]$ as a linear transformation of g_{B_w}: for each $t \in [x_w, x_w + 2^{-2n}]$,

$$f(t) = \langle 2^{-2n} \cdot g_1(2^{2n}(t - x_w)) + x_w,\ 2^{-2n} \cdot g_2(2^{2n}(t - x_w))\rangle,$$

where $\langle g_1(t), g_2(t)\rangle = g_{B_w}(t)$.

Then, we extend f and define it to map the interval $[1, 2]$ piecewise linearly to the three line segments with the following breakpoints: $\langle 1, 0\rangle$, $\langle 1, -1\rangle$, $\langle 0, -1\rangle$, and $\langle 0, 0\rangle$.

It is clear that f is continuous and defines a simple closed curve Γ. Also, f can be proved to be computable in polynomial time since g_B is uniformly polynomial-time computable. We omit the details, which are similar to those in the proof of Theorem 6.5.

Finally, for each w, let $\phi(w) = \langle x_w + 2^{-(2n+1)}, 2^{-(2n+1)}\rangle$. Then, $\phi(w)$ is the image of $\langle 1/2, 1/2\rangle$ under the linear transformation used to define f. It is clear from the transformation that $\delta(\phi(w), \Gamma) \geq 2^{-(2n+2)}$. In addition, $w \in A$ iff $\phi(w)$ is in the interior of the curve Γ. This completes the proof. □

COROLLARY 7.4. *In the following,* (a) ⇒ (b) ⇒ (c).

(a) $FP = \#P$.

(b) *Every simply connected region S with a polynomial-time computable simple curve boundary is P-recognizable.*

(c) $P = UP$.

8. The distance between a point and a curve. Computing the distance between a point $\mathbf{z}$ and a curve Γ is a basic problem in computational complex analysis. In addition, many computational tasks work only when a point $\mathbf{z}$ is bounded away from the curve Γ, e.g., the testing of the aforementioned membership problem and zero problem. What is the complexity of this problem? Or, if Γ is P-computable, does it follow that the function $\text{dist}_\Gamma(\mathbf{z}) = \delta(\mathbf{z}, \Gamma)$ is also P-computable? We observe that this problem is close to the minimization problem, so the complexity bounds for minimization are applicable to the distance problem. The following result on the complexity of the maximization function is from [3].

PROPOSITION 8.1. *The following are equivalent:*

(a) $P = NP$.

(b) *For any P-computable function* $f : [0, 1] \to \mathbf{R}$, *the function* $\max_f(x) = \max\{f(y) : 0 \leq y \leq x\}$ *is P-computable.*

We borrow the ideas of the proof of the above result to characterize the complexity of the distance function.

THEOREM 8.2. *The following are equivalent:*

(a) $P = NP$.

(b) *For any P-computable curve* Γ, *the function* dist_Γ *is also P-computable.*

Proof. For the direction (a) ⇒ (b), we actually prove a stronger statement: for any P-computable curve Γ, there is an oracle machine that computes the function dist_Γ relative to a discrete set A in NP. Assume that Γ is represented by function $f : [0, 1] \to \mathbf{R}^2$, which is computed by an oracle machine M_1 in time p, where p is a polynomial function. Define a set

$$A = \{\langle d_1, d_2, e\rangle : d_1, d_2, e \in \mathbf{D}_n, e \geq 0,\ (\exists d_3 \in \mathbf{D}_{p(n)})\ |M_1^{d_3}(n) - \langle d_1, d_2\rangle| \leq e\}.$$

Then, obviously, $A \in NP$. To compute $\text{dist}_\Gamma(\mathbf{z})$, we first get an approximation $\mathbf{d} = \langle d_1, d_2\rangle$ to $\mathbf{z}$ such that $d_1, d_2 \in \mathbf{D}_{n+2}$ and $|\mathbf{d} - \mathbf{z}| \leq 2^{-(n+2)}$. Next, binary search for a number $e \in \mathbf{D}_{n+2}$ such that $\langle d_1, d_2, e\rangle \in A$ and $\langle d_1, d_2, e - 2^{-(n+2)}\rangle \notin A$. (In the case of $\langle d_1, d_2, 0\rangle \in A$, let $e = 0$.) Then, we can see that $|\text{dist}_\Gamma(\mathbf{z}) - e| \leq 2^{-n}$. First, from $\langle d_1, d_2, e\rangle \in A$, we know that there is a $d_3 \in \mathbf{D}_{p(n+2)} \cap [0, 1]$ such that $|f(d_3) - \mathbf{d}| \leq e + 2^{-(n+2)}$. It follows that $\delta(\mathbf{z}, \Gamma) \leq e + 2^{-(n+2)} + |\mathbf{d} - \mathbf{z}| \leq e + 2^{-(n+1)}$. Conversely, from $\langle d_1, d_2, e - 2^{-(n+2)}\rangle \notin A$, we know that $|f(d_3) - \mathbf{d}| \geq e - 2^{-(n+1)}$ for all $d_3 \in \mathbf{D}_{p(n+2)} \cap [0, 1]$. Therefore, $\delta(\mathbf{d}, \Gamma) \geq e - 2^{-(n+1)} - 2^{-(n+2)}$ and it follows that $\delta(\mathbf{z}, \Gamma) > e - 2^{-n}$.

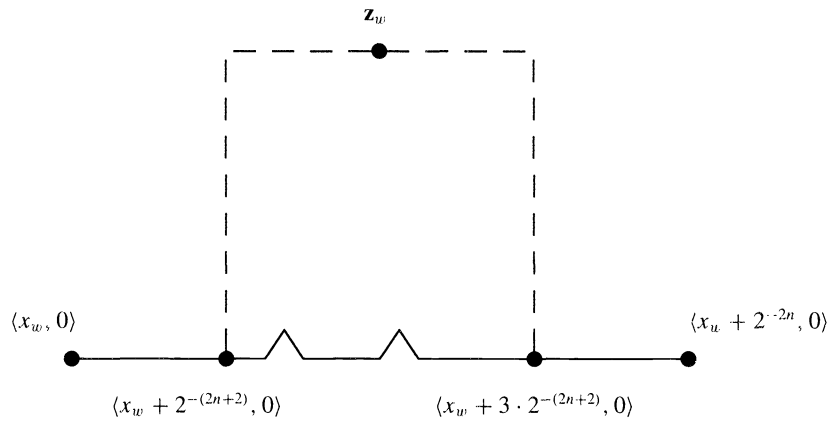


FIG. 3. *Function f on $[x_w, x_w + 2^{-2n}]$ with R_w having exactly 2 strings.*

For the direction (b) $\Rightarrow$ (a), let $A \in NP$. We construct a P-computable function $f : [0, 1] \to \mathbf{R}^2$ that represents a curve Γ such that, if dist_Γ is P-computable, then $A \in P$. First, let $R \in P$ and p be a polynomial function such that for each $w \in \{0, 1\}^*$ of length n, $w \in A$ iff $(\exists u, |u| = p(n))\ \langle w, u\rangle \in R$.

As in Theorem 3.9, we let $a_n = 1 - 2^{-(n-1)}$ and $x_w = a_n + i_w \cdot 2^{-2n}$. We now define function f on interval $[x_w, x_w + 2^{-2n}]$ depending upon set R.

(1) On $[x_w, x_w + 2^{-(2n+2)}]$, f maps the interval linearly to the line segments connecting $\langle x_w, 0\rangle$ to $\langle x_w + 2^{-(2n+2)}, 0\rangle$.

(2) We divide the interval $[x_w + 2^{-(2n+2)}, x_w + 3 \cdot 2^{-(2n+2)}]$ into $2^{p(n)}$ many subintervals of equal size, each corresponding to a string u of length $p(n)$. We let $I_u = [y_u, y_u + 2^{-(2n+1+p(n))}]$ be the interval corresponding to string u. On each I_u, f maps I_u to the horizontal line segment from $\langle y_u, 0\rangle$ to $\langle y_u + 2^{-(2n+1+p(n))}, 0\rangle$ if $\langle w, u\rangle \notin R$. If $\langle w, u\rangle \in R$, then f is piecewise linear on I_u with the following breakpoints: $f(y_u) = \langle y_u, 0\rangle$, $f(y_u + 2^{-(2n+2+p(n))}) = \langle y_u + 2^{-(2n+2+p(n))}, 2^{-(2n+p(n))}\rangle$, and $f(y_u + 2^{(2n+1+p(n))}) = \langle y_u + 2^{-(2n+1+p(n))}, 0\rangle$. That is, f is of the shape $\wedge$ on I_u with height $2^{-(2n+p(n))}$.

(3) On interval $[x_w + 3 \cdot 2^{-(2n+2)}, x_w + 2^{-2n}]$, f maps the interval linearly to the horizontal line segment from $\langle x_w + 3 \cdot 2^{-(2n+2)}, 0\rangle$ to $\langle x_w + 2^{-2n}, 0\rangle$.

We show f on $[x_w, x_w + 2^{-2n}]$ in Fig. 3 for a set R such that $R_w = \{u : \langle w, u\rangle \in R\}$ has two strings. The above definition apparently defines a continuous, P-computable function f.

Now, for each w, let $\mathbf{z}_w = \langle x_w + 2^{-(2n+1)}, 2^{-(2n+1)}\rangle$. Note that the distance between $\mathbf{z}_w$ and $f(t)$ for all $t \leq x_w$ and all $t \geq x_w + 2^{-2n}$ is at least 2^{-2n}. So the distance between $\mathbf{z}_w$ and the curve Γ is the same as the distance between $\mathbf{z}_w$ and the curve $f([x_w, x_w + 2^{-2n}])$. This distance is equal to $2^{-(2n+1)}$ if $w \notin A$ (and hence the curve is flat on this interval), and is $2^{-(2n+1)} - 2^{-(2n+p(n))}$ if $w \in A$.[6] Thus, if dist_Γ is P-computable, then we can determine whether $w \in A$ by computing the $\text{dist}_\Gamma(\mathbf{z}_w)$ correctly within error $2^{-(2n+2+p(n))}$, and hence $A \in P$. □

Next, we extend the above characterization to the problem of determining the distance value between a fixed point and a P-computable curve. That is, if Γ is a P-computable curve and $\mathbf{z}_0$ is a fixed point, is the value $\delta(\mathbf{z}_0, \Gamma)$ always a P-computable real number?

[6] Note that we are using the L_∞-metric on $\mathbf{R}^2$, so the distance is easy to calculate. The function f, nevertheless, could be modified to make the construction work for other metrics, like the L_2-metric, on $\mathbf{R}^2$.

For the maximization problem, it is known that the set of maximum values for P-computable functions from [0, 1] to **R** is exactly the set of *left NP real numbers* [6]. As a corollary, the complexity of maximum values can be characterized as follows. Recall that a *tally set* is a set A over a single alphabet $\{0\}$, and Δ_2^P is the class of sets computable in deterministic polynomial time relative to an oracle in *NP*.

PROPOSITION 8.3 [9]. *In the following,* (a) $\Rightarrow$ (b) $\Rightarrow$ (c).

(a) *All tally sets in* Δ_2^P *are in P.*

(b) *For every P-computable function* $f : [0, 1] \to \mathbf{R}$, *the maximum value* $x = \max\{f(y) : 0 \leq y \leq 1\}$ *is P-computable.*

(c) *All tally sets in NP are in P.*

It is interesting to note that condition (c) above is equivalent to the condition that the class *NEXP* of nondeterministic exponential time $(2^{n^{O(1)}})$ computable sets collapses to the class *EXP* of deterministic exponential time computable sets.

The above result on the maximum values can be applied to characterize the complexity of distance values.

THEOREM 8.4. *A real number x is right NP iff there exists a P-computable curve* Γ *such that* $x = \delta(\langle 0, 0\rangle, \Gamma)$.

Proof. To see that dist_Γ must be a right *NP* real number, we simply observe that the set $B = \{e : e \in \mathbf{D}_n, \langle 0, 0, e\rangle \in A\}$ is a right cut of dist_Γ, where A is the set defined in the proof of direction (a) $\Rightarrow$ (b) of Theorem 8.2.

Conversely, we define a simple reduction from the maximum values to distance values. For any P-computable function $f : [0, 1] \to [0, 1]$, let $g : [0, 1] \to \mathbf{R}^2$ be defined by $g(x) = \langle x - 1/2, f(x) - 2\rangle$; i.e., the curve defined by g is the graph of f moved left by $1/2$ and downward by 2. We can see that the distance between the origin $\langle 0, 0\rangle$ and the curve defined by g is exactly $2 - \max\{f(y) : 0 \leq y \leq 1\}$. Thus, any right *NP* real number x could be made equal to the distance between a P-computable curve and the origin $\langle 0, 0\rangle$. □

The above immediately implies the following complexity characterization of distance values.

COROLLARY 8.5. *In the following,* (a) $\Rightarrow$ (b) $\Rightarrow$ (c).

(a) *All tally sets in* Δ_2^P *are in P.*

(b) *For every P-computable function* $f : [0, 1] \to \mathbf{R}^2$ *that represents a curve* Γ *and every P-computable point* $\mathbf{z} \in \mathbf{R}^2$, *the distance* $\delta(\mathbf{z}, \Gamma)$ *is a P-computable real number.*

(c) *All tally sets in NP are in P.*

9. The area of a region. In this section, we consider the problem of computing the area of a region, given either a P-approximation machine M or a boundary Γ as the representation. Assume that a simply connected region S is P-approximable; then, intuitively, the area can be computed by the straightforward sampling method, which can be done in polynomial time relative to an oracle in $\#P$. Indeed, Ko [7] has shown that the computation of the measure of a P-approximable one-dimensional set $S \subseteq [0, 1]$ can be done in polynomial time if $FP = \#P$. This proof can be transformed to two-dimensional P-approximable sets. Conversely, Friedman [3] has shown that if $FP \neq \#P$, then there exists a P-computable function $f : [0, 1] \to \mathbf{R}$ such that its integration function $g(x) = \int_0^x f(t)dt$ is not P-computable. This result can be easily extended to two-dimensional regions. In the following, FP_1 (and $\#P_1$) denotes the class of functions ϕ in FP (and $\#P$, respectively), whose inputs are strings over a singleton alphabet $\{0\}$.

THEOREM 9.1. *The following are equivalent:*

(a) $FP_1 = \#P_1$.

(b) *For any P-approximable set* $S \subseteq [0, 1]^2$, *the measure of set S is polynomial-time computable.*

(c) *For any simply connected region S that is P-recognizable and has a P-computable, rectifiable boundary, the measure of S is P-computable.*

Proof. (a) $\Rightarrow$ (b). Assume that $S \subseteq [0, 1]^2$ is P-approximable. By Theorem 3.3, there is a uniformly polynomial-time computable sequence $\{S_n\}$ of finite union of rectangles such that $\mu^*(S_n \Delta S) \leq 2^{-n}$. Let p be a polynomial such that the corners of the rectangles in S_n are in $\mathbf{D}_{p(n)} \times \mathbf{D}_{p(n)}$. Then, for each point $\mathbf{d} = \langle d_1, d_2 \rangle \in \mathbf{D}_{p(n)+1} \times \mathbf{D}_{p(n)+1}$, $\mathbf{d}$ is an interior point of S_n iff the square $N(\mathbf{d}; 2^{-(p(n)+1)})$ is contained in S_n. The measure of S_n thus is equal to $2^{-(2p(n)+2)}$ times the size of the following set:

$$A_n = \{\mathbf{d} = \langle d_1, d_2 \rangle : d_1, d_2 \in \mathbf{D}_{p(n)+1} \cap (0, 1), d \text{ is an interior point of } S_n\}.$$

Since S_n is uniformly polynomial-time computable, we can determine in polynomial time whether a point $\mathbf{d} \in \mathbf{D}_{p(n)+1} \times \mathbf{D}_{p(n)+1}$ is an interior point of S_n. Therefore, the function that maps the input 0^n to the size of A_n is in $\#P_1$. This implies that we can compute the size of A_n, and hence the measure of S_n, in polynomial time relative to an oracle in $\#P_1$. The proof is completed by observing the fact that $\mu^*(S_n \Delta S) \leq 2^{-n}$.

(b) $\Rightarrow$ (c). The proof is immediate from Theorem 3.7(a).

(c) $\Rightarrow$ (a). Let $G : \{0\}^* \to \mathbf{N}$ be a function in $\#P_1$ that is not in FP_1. Then, there exists a polynomial p such that $G(0^n) \leq 2^{p(n)}$. We may assume that $p(n+1) > p(n)$ for all $n > 0$. We first claim that the real number

$$a = \sum_{n=1}^{\infty} G(0^n) \cdot 2^{-2(p(n)^2+n)}$$

is not polynomial-time computable.

Proof. Because $p(n+1) > p(n)$, we have $2p(n+1)^2 - p(n+1) > p(n)^2$. This implies that

$$G(0^{n+1}) \cdot 2^{-2(p(n+1)^2+n+1)} \leq 2^{-(2p(n+1)^2 - p(n+1)+2n+2)} < 2^{-(2p(n)^2+2n+2)}$$

or, equivalently, in the binary expansion of number a, the bits from the $(2p(n-1)^2+2n+1)$st bit to the right of the binary point to the $(2p(n)^2+2n)$th bit precisely encode the binary expansion of the integer $G(0^n)$. In addition, the $(2p(n)^2 + 2n + 1)$st and $(2p(n)^2 + 2n + 2)$nd bits of the binary expansion of a must be 0. These properties then imply that a is not polynomial-time computable. To be more precise, we note that if d is a dyadic rational in $\mathbf{D}_{2p(n)^2+2n+2}$ such that $|d - a| \leq 2^{-(2p(n)^2+2n+2)}$, then the first $2p(n)^2 + 2n$ of the binary expansion of d must agree with those of a, since the error in the approximation of a by d cannot propagate through the $(2p(n)^2 + 2n + 1)$st bits. Therefore, any polynomial-time algorithm for a could be used to extract the first $2p(n)^2 + 2n$ bits of a and obtain the value $G(0^n)$. That would be a contradiction. □

Next, by Proposition 6.2(b) we know that there exists a set $A \subseteq \{0, 1\}^*$ in P such that $G(0^n)$ is equal to the size of the set $A_n = \{u \in \{0, 1\}^* : \ell(u) = p(n), u \in A\}$. In the following algorithm, we define from set A a function $f : [0, 2] \to \mathbf{R}^2$ that defines a rectifiable, simple, closed curve Γ such that the measure of the interior of Γ is equal to $1 + a$.

> ALGORITHM FOR f. (1) The function f maps $[1, 2]$ piecewise linearly to three line segments defined by the following four points: $\langle 1, 0 \rangle$, $\langle 1, -1 \rangle$, $\langle 0, -1 \rangle$, $\langle 0, 0 \rangle$.
> (2) For each $n > 0$, let $a_n = 1 - 2^{-(n-1)}$. Divide the interval $[a_n, a_{n+1}]$ into $2^{p(n)}$ many subintervals $[t_k, t_{k+1}]$, $0 \leq k \leq 2^{p(n)} - 1$, where $t_k = a_n + k \cdot 2^{-(p(n)+n)}$. For each k, $0 \leq k \leq 2^{p(n)} - 1$, let u_k be the kth string in

$\{0,1\}^{p(n)}$. If $u_k \notin A_n$, then let $f(t) = \langle t, 0\rangle$ for all $t \in [t_k, t_{k+1}]$. If $u_k \in A_n$, then f maps the interval $[t_k, t_{k+1}]$ linearly to four consecutive line segments defined by the following five breakpoints: $\langle t_k, 0\rangle$, $\langle t_k, 2^{-(p(n)^2+n)}\rangle$, $\langle t_k + 2^{-(p(n)^2+n)}, 2^{-(p(n)^2+n)}\rangle$, $\langle t_k + 2^{-(p(n)^2+n)}, 0\rangle$, $\langle t_{k+1}, 0\rangle$. In other words, the curve defined by f on $[t_k, t_{k+1}]$ forms an open square $\sqcap$, of size $2^{-(p(n)^2+n)} \times 2^{-(p(n)^2+n)}$, above the line $y = 0$.

It is easy to verify that f is P-computable. Furthermore, the interior S of Γ is P-recognizable, since for any point $\langle d_1, d_2\rangle \in \mathbf{D} \times \mathbf{D}$ with $a_n \le d_1 \le a_{n+1}$ and $0 \le d_2 < 2^{-(p(n)^2+n)}$, $\langle d_1, d_2\rangle \in S$ iff $u_k \in A_n$ and $t_k < d_1 < t_k + 2^{-(p(n)^2+n)}$, where k is the unique integer such that $d_1 \in [t_k, t_{k+1})$.

Next we show that the curve Γ defined by f on $[0, 1]$ is of finite length. We note that for each n and each k, $0 \le k \le 2^{p(n)} - 1$, the function f maps $[t_k, t_{k+1}]$ to a line segment of length $2^{-(p(n)+n)}$, or to four line segments of length $2^{-(p(n)+n)} + 2 \cdot 2^{-(p(n)^2+n)}$. In either case, the total length is at most $2^{-(p(n)+n-1)}$. Thus, the total length of the curve defined by f on $[a_n, a_{n+1}]$ is at most $2^{-(n-1)}$. Thus, the total length of Γ is at most $3 + \sum_{n=1}^{\infty} 2^{-(n-1)} = 5$.

Finally, we observe that the measure of the interior S of the curve Γ is equal to that of the square $[0, 1] \times [-1, 1]$ plus the small squares above the line $y = 0$. On each interval $[a_n, a_{n+1}]$, we have defined exactly $G(0^n)$ small squares above the line $y = 0$ each of size $2^{-(p(n)^2+n)} \times 2^{-(p(n)^2+n)}$. Therefore, the total measure of all these squares for all $n > 0$ is equal to $a = \sum_{n=1}^{\infty} G(0^n) \cdot 2^{-2(p(n)^2+n)}$. This implies that the measure of the interior of Γ is $a + 1$, which is not a polynomial-time computable real number. □

Remark. The above rectifiability condition on the boundary of S in part (c) of Theorem 9.1 is necessary, since Ko [10] has constructed a P-computable function $f : [0, 1] \to \mathbf{R}^2$ that represents a simple, closed, nonrectifiable curve Γ whose interior S is P-recognizable but has a nonrecursive measure. This result suggests that when simply connected regions are represented by their boundary representation, the rectifiability of the boundary curve, in addition to the P-computability of the curve, is an important factor that might affect the complexity of the region.

REFERENCES

[1] J. Balcázar, J. Diaz, and J. Gabarró, *Structural Complexity I*, Springer-Verlag, Berlin, 1988.

[2] L. Blum, M. Shub, and S. Smale, *On a theory of computation and complexity over the real numbers; NP completeness, recursive functions and universal machines*, Bull. Amer. Math. Soc. (N.S.), 21 (1989), pp. 1–46.

[3] H. Friedman, *On the computational complexity of maximization and integration*, Adv. Math., 53 (1984), pp. 80–98.

[4] F. Green, J. Köbler, and J. Torán, *The power of the middle bit*, in Proc. 7th Structure in Complexity Theory Conference, Boston, MA, IEEE Computer Society, 1992, pp. 111–117.

[5] P. Henrici, *Applied and Computational Complex Analysis*, Vol. 1, John Wiley, New York, 1974.

[6] K. Ko, *The maximum value problem and NP real numbers*, J. Comput. System Sci., 24 (1982), pp. 15–31.

[7] ———, *Approximation to measurable functions and its relation to probabilistic computation*, Ann. Pure Appl. Logic, 30 (1986), pp. 173–200.

[8] ———, *Computational complexity of roots of real functions*, in Proc. 30th IEEE Symposium on Foundations of Computer Science, Research Triangle Park, NC, IEEE Computer Society, 1989, pp. 204–209.

[9] ———, *Complexity Theory of Real Functions*, Birkhäuser, Boston, 1991.

[10] ———, *A polynomial-time computable curve whose interior has a nonrecursive measure*, Theoret. Comput. Sci., to appear.

[11] K. Ko and H. Friedman, *Computational complexity of real functions*, Theoret. Comput. Sci., 20 (1982), pp. 323–352.

[12] L. Lovász, *An Algorithmic Theory of Numbers, Graphs and Convexity*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1986.

[13] C. A. NEFF, *Specified precision polynomial root isolation is in NC*, in Proc. 31st IEEE Symposium on Foundations of Computer Science, St. Louis, MO, IEEE Computer Society, 1990, pp. 152–162.

[14] A. NERODE AND W. HWANG, *Applications of pure recursion theory in recursive analysis*, Acta Math. Sinica, 28 (1985), pp. 625–636. (In Chinese.)

[15] C. H. PAPADIMITRIOU AND J. TSITSIKLIS, *Intractable problems in control theory*, SIAM J. Control Optim., 24 (1986), pp. 639–654.

[16] M. POUR-EL AND I. RICHARDS, *Computability in Analysis and Physics*, Springer-Verlag, Berlin, 1989.

[17] K. REGAN AND T. SCHWENTICK, *On the power of one bit of a #P function*, Proc. Annual Italian Conference on Theoretical Computer Science, World Scientific, Singapore, 1992, pp. 317–329.

[18] A. SCHÖNHAGE, *The fundamental theorem of algebra in terms of computational complexity*, Mathematisches Institut der Universität Tübingen, West Germany, 1982, preprint.

[19] S. SMALE, *The fundamental theorem of algebra and complexity theory*, Bull. Amer. Math. Soc., 4 (1981), pp. 1–36.

[20] J. F. TRAUB, G. W. WASILKOWSKI, AND H. WOŹNIAKOWSKI, *Information-Based Complexity*, Academic Press, New York, 1988.

SIAM J. COMPUT.
Vol. 24, No. 5, pp. 948–969, October 1995

NEARLY OPTIMAL ALGORITHMS FOR CANONICAL MATRIX FORMS*

MARK GIESBRECHT†

Abstract. A Las-Vegas-type probabilistic algorithm is presented for finding the Frobenius canonical form of an $n \times n$ matrix T over any field K. The algorithm requires $O^{\sim}(\mathrm{MM}(n)) = \mathrm{MM}(n) \cdot (\log n)^{O(1)}$ operations in K, where $O(\mathrm{MM}(n))$ operations in K are sufficient to multiply two $n \times n$ matrices over K. This nearly matches the lower bound of $\Omega(\mathrm{MM}(n))$ operations in K for this problem and improves on the $O(n^4)$ operations in K required by the previous best-known algorithms. A fast parallel implementation of the algorithm is also demonstrated for the Frobenius form, which is processor-efficient on a PRAM. As an application we give an algorithm to evaluate a polynomial $g \in \mathsf{K}[x]$ at T which requires only $O^{\sim}(\mathrm{MM}(n))$ operations in K when $\deg g \leq n^2$. Other applications include sequential and parallel algorithms for computing the minimal and characteristic polynomials of a matrix, the rational Jordan form of a matrix (for testing whether two matrices are similar), and matrix powering, which are substantially faster than those previously known.

Key words. Frobenius form, Jordan form, evaluating polynomials at matrices, matrix powering, matrix multiplication, processor-efficient parallel algorithms

AMS subject classifications. 15-04, 15A21, 65Y05, 65Y20

1. Introduction.

Computing a canonical or normal form of an $n \times n$ matrix T over any field K is a classical mathematical problem with many practical applications. A fundamental theorem of linear algebra states that any $T \in \mathsf{K}^{n \times n}$ is similar to a unique matrix $S \in \mathsf{K}^{n \times n}$ of the block diagonal form

$$S = \mathrm{diag}(C_{f_1}, C_{f_2}, \ldots, C_{f_k}) = \begin{pmatrix} C_{f_1} & & & 0 \\ & C_{f_2} & & \\ & & \ddots & \\ 0 & & & C_{f_k} \end{pmatrix} \in \mathsf{K}^{n \times n}, \tag{1.1}$$

where each C_{f_i} is the companion matrix of some monic $f_i \in \mathsf{K}[x]$ for $1 \leq i \leq k$, and $f_i \mid f_{i-1}$ for $2 \leq i \leq k$ (we write $T \sim S$ to denote similarity). Recall that the companion matrix C_g of a monic $g = \sum_{0 \leq j \leq r} b_i x^i \in \mathsf{K}[x]$ has the form

$$C_g = \begin{pmatrix} 0 & 0 & & & -b_0 \\ 1 & 0 & & & -b_1 \\ & 1 & & 0 & \vdots \\ & & \ddots & & \vdots \\ & 0 & & \ddots & \\ & & & 1 & -b_{r-1} \end{pmatrix} \in \mathsf{K}^{r \times r}.$$

A matrix S with these properties, called the *Frobenius form* of T, always exists and is unique and *rational*; i.e., it remains the same even if the entries of T are allowed to lie in

*Received by the editors July 26, 1993; accepted for publication (in revised form) March 22, 1994. This research was supported in part by Natural Sciences and Engineering Research Council of Canada research grant OGP0155376.

†Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, R3T 2N2, Canada (mwg@cs.umanitoba.ca).

an algebraic closure of K. The polynomials $f_1, \ldots, f_k \in \mathsf{K}[x]$ are the *invariant factors* of T, and the first of these f_1 is the *minimal polynomial* of T; that is, f_1 is the polynomial of smallest degree in $\mathsf{K}[x] \setminus \{0\}$ such that $f_1(T) = 0$. The product $f_1 \cdots f_k$ is the characteristic polynomial of T. Since T is similar to S, by definition there exists an invertible $U \in \mathsf{K}^{n \times n}$ such that $U^{-1}TU = S$. Our approach to finding the Frobenius form is to find such a matrix U, from which we find S. Two excellent references for the background information are Gantmacher (1990), Chapter 7, and Hoffman and Kunze (1971), Chapter 7.

In §§2–5 we link the cost of computing the Frobenius form to the cost of multiplying two matrices. Specifically, if $\mathrm{MM}(n)$ operations in K are sufficient to multiply two $n \times n$ matrices over a field K, we show that $O^{\sim}(\mathrm{MM}(n)) = \mathrm{MM}(n) \cdot (\log n)^{O(1)}$ operations in K suffice to compute the Frobenius form of an $n \times n$ matrix over K. This algorithm is of the Las Vegas type: it is allowed to choose elements randomly and uniformly from a finite subset of K at unit cost, and with probability at least $1/4$ it returns the correct answer, otherwise it reports failure. An incorrect answer is never returned. Our algorithm only works as stated when $\#\mathsf{K} \geq n^2$. When K has $q < n^2$ elements, we embed K in a field F of degree $O(\log n)$ over K which does possess n^2 elements. The Frobenius form of T lies in $\mathsf{K}^{n \times n}$ since it is rational, but $U \in \mathsf{F}^{n \times n}$ may have entries in $\mathsf{F} \setminus \mathsf{K}$. In this case the running time of our algorithms is multiplied by a small power of $\log_q n$. Details are presented in §5. In §6 we show how to implement our algorithm for the Frobenius form in a processor-efficient manner on a PRAM. As a by-product we obtain the first processor-efficient parallel algorithm for the characteristic polynomial of a matrix.

The best previously known algorithms for finding the Frobenius form of a matrix in $\mathsf{K}^{n \times n}$, by Ozello (1987) and Lüneburg (1987), require $O(n^4)$ operations in K (see also Kannan and Bachem (1979), Kannan (1985), and Augot and Camion (1993)). Kaltofen, Krishnamoorthy, and Saunders (1987), (1990) present probabilistic algorithms for finding the Frobenius form of a matrix in the parallel complexity class RNC^2. They achieve their result through the use of a more general algorithm to compute the Smith normal form of a polynomial matrix. From the Smith normal form of the matrix $\lambda I - T \in \mathsf{K}[\lambda]^{n \times n}$ (where λ is an indeterminate) the Frobenius form S of T can be derived easily. A (deterministic) NC^2 algorithm for computing the Frobenius form is demonstrated by Villard (1994).

Our algorithm for computing the Frobenius form is nearly optimal in that there is a lower bound for the problem of $\Omega(\mathrm{MM}(n))$ operations in K. If we can compute the Frobenius form, then we can find the characteristic polynomial $f \in \mathsf{K}[x]$ of T with $O(n^2)$ additional operations in K. The determinant of T is $f(0)$, and it is shown in Baur and Strassen (1982) that computing the determinant requires $\Omega(\mathrm{MM}(n))$ operations in K, whence computing the Frobenius form of T requires $\Omega(\mathrm{MM}(n))$ operations in K (one must be careful of the model of computation here as Baur and Strassen's result is for the arithmetic circuit model; see Giesbrecht (1993), §1.1 for details).

We obtain fast algorithms for a number of interesting problems as applications of our algorithm for computing the Frobenius form. One of the most striking results is for the problem of evaluating a polynomial at a matrix. In §7 we show that a polynomial $g \in \mathsf{K}[x]$ of degree r can be evaluated at any matrix $T \in \mathsf{K}^{n \times n}$ with $O^{\sim}(\mathrm{MM}(n) + r)$ operations in K. We also demonstrate a lower bound for evaluating a fixed nonlinear polynomial at a matrix of $\Omega(\mathrm{MM}(n))$ operations in K, so our algorithm is, in fact, nearly optimal. The algorithm we present here improves upon the previously fastest algorithm of Paterson and Stockmeyer (1973), which requires $O(\mathrm{MM}(n)\sqrt{r})$ operations in K. More generally, we show that an arithmetic circuit or straight-line program can be evaluated at a matrix in nearly optimal time sequentially and processor-efficiently in parallel. In brief, the Frobenius form provides a mechanism to transform the problem of evaluating a polynomial in $\mathsf{K}[x]$ at a matrix in

$\mathsf{K}^{n\times n}$ from the multiplicative semigroup of $\mathsf{K}^{n\times n}$ to the multiplicative semigroup of a modular polynomial ring, where computation can be performed much more quickly.

As noted above, the companion matrix of the minimal polynomial forms the first block in the Frobenius form. Also, any two similar matrices have the same Frobenius form. Thus, our algorithm for computing the Frobenius form yields Las Vegas algorithms to determine the similarity of any two matrices in $\mathsf{K}^{n\times n}$ and find the minimal polynomial of a matrix, which require $O\tilde{}(\mathrm{MM}(n))$ operations in K. Our algorithm for computing the minimal polynomial is also nearly optimal: it is shown by Wiedemann (see also Kaltofen (1992)) that if we can compute the minimal polynomial of an $n \times n$ matrix over a field K with $t(n)$ operations in K, then there is a Las Vegas algorithm to compute the determinant of any matrix in $\mathsf{K}^{n\times n}$ for which $O(t(n))$ operations in K are sufficient (see Giesbrecht (1993) for details). The best previously known algorithm for computing the minimal polynomial of an $n \times n$ matrix is the Monte Carlo algorithm of Wiedemann (1986), and this algorithm requires an expected $O(n^3)$ operations (a Monte-Carlo-type algorithm has the ability to select random elements uniformly from a finite subset of K, and with constant probability it returns the correct answer, but with some controllably small probability it may return an incorrect answer). To determine similarity, the best previously known sequential algorithms are the Frobenius form algorithms of Lüneburg (1987) and Ozello (1987), which require $O(n^4)$ field operations. An algorithm by Zalcstein and Garzon (1987), based on a very different method, runs in the parallel complexity class NC^2.

The Jordan normal form is probably the most commonly encountered of all canonical matrix forms, and also one of the most difficult to compute. In particular, it requires that we determine both the geometric structure of the matrix (as captured in the Frobenius form) and the factorization of the minimal polynomial of the matrix into linear factors. In §8 we introduce the rational Jordan form as a slight generalization of the usual Jordan form. The rational Jordan form always exists and coincides with the usual Jordan form whenever that form exists. We show that computing the rational Jordan form of an $n \times n$ matrix over any field K can be accomplished by a Las-Vegas-type algorithm with an expected number of $O\tilde{}(\mathrm{MM}(n))$ operations in K, given the complete factorization (into irreducible factors in $\mathsf{K}[x]$) of the minimal polynomial of that matrix. A simple lower bound of $\Omega(\mathrm{MM}(n))$ operations in K is shown for this problem. The requirement of a complete factorization of the minimal polynomial is also necessary, given that the complete factorization of any polynomial can be read off the rational Jordan form of the companion matrix of that polynomial.

Computational model and complexity assumptions. The algorithms presented in this paper are generally given for both sequential and parallel models of computation. For the sequential algorithms we employ the arithmetic RAM, described more formally in von zur Gathen (1993). Informally, this is just a standard (Boolean) RAM (see Aho, Hopcroft, and Ullman (1974), §1.2) with an additional memory for holding elements of some field K. The instructions of the usual RAM are supplemented with instructions for basic field operations in K ($+$, $\times$, $-$, $\div$), as well as a test for the zero element in K and appropriate input and output instructions for elements in K. We report the cost of our algorithms as the number of operations in K required as a function of the number of elements of K in the input.

For parallel algorithms we employ the arithmetic PRAM model over a field K (see Karp and Ramachandran (1990)). This is a collection of arithmetic RAMs (over K) communicating by means of a set of shared memory cells. All accesses to global (and local) memory require unit time. The time $t(n)$ required by an arithmetic PRAM algorithm is the maximum of the number of field operations required by each of the component arithmetic RAMs in the PRAM on input size n. Also of interest is $p(n)$, the largest number of processors involved in the computation on any input of size n, as is the work $w(n) = t(n) \cdot p(n)$. It is useful to compare

the work required by an algorithm running on a PRAM with the time required by an optimal sequential algorithm. Assume that any sequential algorithm for some problem requires time $\Omega(t(n))$ on input size n. A PRAM algorithm for this problem is called *processor-efficient* if it requires time $(\log n)^{O(1)}$ when run on $t(n) \cdot (\log n)^{O(1)}$ processors. That is, it is a fast parallel algorithm for which the work is within a polylogarithmic factor of the optimal.

It is occasionally convenient, especially in summarizing results, to ignore logarithmic factors using the "soft O" notation: for any $g, h: \mathbb{R}_{>0} \to \mathbb{R}_{>0}$, $g = \tilde{O}(h)$ if and only if there exists a constant $k \geq 0$ such that $g = O(h(\log h)^k)$.

We isolate the cost of algorithms for polynomial arithmetic and linear algebra as named functions (M and MM, respectively) of their input sizes when their costs appear in the cost of some algorithm using these operations. Over a ring F, we assume $O(\mathrm{M}_{\mathsf{F}}(n))$ operations in F are sufficient to multiply two polynomials in $\mathsf{F}[x]$ of degree at most n. The fast integer multiplication algorithms of Schönhage and Strassen (1971) can be recast as polynomial multiplication algorithms (see Schönhage (1977) and Nussbaumer (1980)) and allow us to choose $\mathrm{M}_{\mathsf{F}}(n) = n \log n \log\log n$ when F is a field. More generally, for any ring F, the algorithm of Cantor and Kaltofen (1991) allows us to choose $\mathrm{M}_{\mathsf{F}}(n) = n \log n \log\log n$. If K is a field and $f, g \in \mathsf{K}[x]$ have degree at most n, we can compute the division with a remainder of f by g; that is, we can find $Q, R \in \mathsf{K}[x]$ such that $f = Qg + R$ with $R = 0$ or $\deg R < \deg g$ in $O(\mathrm{M}_{\mathsf{F}}(n))$ operations in K. We can compute $\gcd(f, g)$ with $O(\mathrm{M}_{\mathsf{F}}(n) \log n)$ operations in K (see Aho, Hopcroft, and Ullman (1974), §8.9). Generally, we will simply write $\mathrm{M}(n)$ for $\mathrm{M}_{\mathsf{F}}(n)$ when F is clear from the context.

For any field K, let $\mathsf{K}^{n\times n}$ be the ring of $n \times n$ matrices with entries in K. We assume that $O(\mathrm{MM}(n))$ operations in K are sufficient to multiply two matrices in $\mathsf{K}^{n\times n}$. Currently, the asymptotically best algorithm for matrix multiplication is by Coppersmith and Winograd (1990) with $\mathrm{MM}(n) = n^{2.376}$. For convenience we assume throughout this paper that $\mathrm{MM}(n) = \Omega(n^{2+\epsilon})$ for some $\epsilon > 0$. $O(\mathrm{MM}(n))$ operations in K are also sufficient to compute the rank and determinant of an $n \times n$ matrix over a field K, as well as invert a nonsingular matrix in $\mathsf{K}^{n\times n}$. Also, we can solve a system of n linear equations in n unknowns over K (which may be singular) with $O(\mathrm{MM}(n))$ operations in K (Bunch and Hopcroft (1974)). Corresponding parallel algorithms for the polynomial and matrix operations above are discussed in §6.

2. Finding a modular cyclic decomposition. The Frobenius normal form of the matrix $T \in \mathsf{K}^{n\times n}$ corresponds to a decomposition of the vector space $\mathsf{K}^{n\times 1}$, called the *cyclic decomposition* of $\mathsf{K}^{n\times 1}$ with respect to T. It is convenient to simply consider the linear map T as a K-endomorphism of a finite dimensional vector space V over K (i.e., $T \in \mathrm{End}_{\mathsf{K}} V$), ignoring for now its representation as a matrix (in the case of $T \in \mathsf{K}^{n\times n}$ we have $V = \mathsf{K}^{n\times 1}$). V is a $\mathsf{K}[x]$-module as follows: any $g = \sum_{0\leq i\leq r} b_i x^i \in \mathsf{K}[x]$ acts on V as the endomorphism $\sum_{0\leq i\leq r} b_i T^i \in \mathrm{End}_{\mathsf{K}} V$. We write $f \circ v = f(T)v \in V$. An important role is played in this theory by polynomials which annihilate (i.e., map to $0 \in V$) vectors and subspaces of V. For any $v \in V$, define the *annihilator* $\mathrm{Ann}(T; v) \subseteq \mathsf{K}[x]$ of V as the ideal in $\mathsf{K}[x]$ of all $f \in \mathsf{K}[x]$ such that $f \circ v = 0$. Since $\mathsf{K}[x]$ is a principal ideal domain, $\mathrm{Ann}(T; v)$ is generated by a unique monic polynomial in $\mathsf{K}[x]$, the *minimal polynomial* $\min(T; v) \in \mathsf{K}[x]$ of the vector $v \in V$. The subspace $\mathrm{Orb}(T; v) \subseteq V$ spanned by $v, Tv, T^2v, \ldots \in V$ is the *cyclic subspace* of V generated by v. $\mathrm{Orb}(T; v)$ is T-invariant, that is, $Tw \in \mathrm{Orb}(T; v)$ for all $w \in \mathrm{Orb}(T; v)$, and $\dim \mathrm{Orb}(T; w) = \deg(\min(T; v))$. For any subspace $W \subseteq V$, the set of polynomials $\mathrm{Ann}(T; W) \subseteq \mathsf{K}[x]$ which annihilate *every* vector in W is also an ideal in $\mathsf{K}[x]$, called the *annihilator* of W. As an ideal, it too is generated by a unique monic polynomial, the *minimal polynomial* $\min(T; W) \in \mathsf{K}[x]$ of T on W.

The following very important theorem is shown in Hoffman and Kunze (1971), §7.2, Theorem 3, and Gantmacher (1990), §7.5.

FACT 2.1 (cyclic decomposition theorem). *Let V be a finite dimensional vector space over a field K and $T \in \mathrm{End}_{\mathsf{K}} V$. There exist vectors $v_1, v_2, \ldots, v_k \in V$ such that*

$$V = V_1 \oplus V_2 \oplus \cdots \oplus V_k,$$

where $V_i = \mathrm{Orb}(T; v_i)$, and

$$\min(T; v_i) = \min(T; V_i \oplus \cdots \oplus V_k) \in \mathsf{K}[x]$$

for $1 \leq i \leq k$.

Given a cyclic decomposition of V under T as $V = V_1 \oplus \cdots \oplus V_k$, where $V_i = \mathrm{Orb}(T; v_i)$ for some $v_i \in V$ for $1 \leq i \leq k$, $f_i = \min(T; v_i) \in \mathsf{K}[x]$ is the *ith invariant factor* of T for $1 \leq i \leq k$. Recall that the companion matrices of the invariant factors are found, in order, along the diagonal of the Frobenius form of T.

We once again consider T as a matrix in $\mathsf{K}^{n \times n}$, and the relationship between the cyclic decomposition of $\mathsf{K}^{n \times 1}$ under T and T's Frobenius form. Suppose $\deg f_i = d_i$ for $1 \leq i \leq k$. The matrix

$$U = \left[v_1 \,\middle|\, Tv_1 \,\middle|\, T^2 v_1 \,\middle|\, \cdots \,\middle|\, T^{d_1 - 1} v_1 \,\middle|\, \cdots \,\middle|\, v_k \,\middle|\, Tv_k \,\middle|\, \cdots \,\middle|\, T^{d_k - 1} v_k \right] \in \mathsf{K}^{n \times n} \tag{2.1}$$

has the property that $S = U^{-1} T U$ is in Frobenius normal form (see, for example, Hoffman and Kunze (1971), §7.5).

Our algorithm for finding the Frobenius normal form S of a matrix $T \in \mathsf{K}^{n \times n}$ proceeds by constructing the matrix U as in (2.1), and then computing $S = U^{-1} T U$. To do this we must somehow find the invariant factors $f_1, \ldots, f_k \in \mathsf{K}[x]$ and an associated set of vectors $v_1, \ldots, v_k \in V$ generating the cyclic components. Unfortunately, such vectors are very rare, and the approach of choosing them randomly will fail miserably. Ozello (1987) and Lüneburg (1987) construct them deterministically, but using more time than we allow ourselves. Our approach will be to choose randomly first a list of vectors $w_1, \ldots, w_k \in V$. With high probability these generate a "modular cyclic decomposition" of V (which we define later), from which the invariant factors and the Frobenius form S of T can be determined. The vectors $w_1, \ldots, w_k$ are then "purified" to obtain vectors $v_1, \ldots, v_k$ generating the components of the cyclic decomposition. Our construction follows approximately the geometric proof of the cyclic decomposition theorem by Gantmacher (1990), Chapter 7 (see also Hoffman and Kunze (1971), §7.2).

A weaker form of the cyclic decomposition theorem can be stated in terms of modular T-invariant vector spaces. If W is a T-invariant subspace of V then the space V/W, or V modulo W, is a K-vector space of dimension $\dim V - \dim W$. For $w_1, w_2 \in V$ we denote by $w_1 + W$ and $w_2 + W$ the images of w_1 and w_2 in V/W, respectively, and say $w_1 \equiv w_2 \bmod W$ when $w_1 - w_2 \in W$. Since W is T-invariant, V/W has a $\mathsf{K}[x]$-module structure induced by the action of $\mathsf{K}[x]$ on V. Let $T_w : V/W \to V/W$ be T reduced modulo W, carrying $v + W$ to $Tv + W$ for any $v \in V$. Then V/W is a $\mathsf{K}[x]$-module, where $f \in \mathsf{K}[x]$ acts as $f(T_w) : V/W \to V/W$, and we write $f \circ (v + W) = f(T_w)(v + W) = f(T)v + W$. Thus, $\min(T_w; v + W) \in \mathsf{K}[x]$ is the minimal polynomial of $v + W \in V/W$ in $\mathsf{K}[x]$, that is, the monic $f \in \mathsf{K}[x] \setminus \{0\}$ of minimal degree such that $f \circ (v + W) \equiv 0 \bmod W$.

An important role is played in the decomposition theory of a vector space by *maximal* vectors $v \in V$, those satisfying $\min(T; v) = \min(T; V)$. A proof of the following lemma can be found in Gantmacher (1990), §7.4.

FACT 2.2. *Let V be any vector space over K and $T : V \to V$ be a linear map with invariant factors $f_1, \ldots, f_k \in \mathsf{K}[x]$, where $f_i \mid f_{i-1}$ for $2 \leq i \leq k$. There exists a maximal $v_1 \in V$ such*

that $\min(T; v_1) = \min(T; V) = f_1$, *and for all such* v_1 *there exists a* T*-invariant subspace* $V' \subseteq V$ *such that* $V = \mathrm{Orb}(T; v_1) \oplus V'$. *Furthermore, the invariant factors of* T *on* V' *are* $f_2, \ldots, f_k$.

The utility of this fact comes in the realization that, for v_1 and V' as above, V' is isomorphic as a $\mathsf{K}[x]$-module to $V/\mathrm{Orb}(T; v_1)$. This follows since every element $w \in V$ can be written uniquely as $w = \bar{w} + w'$ for some $\bar{w} \in \mathrm{Orb}(T; v_1)$ and $w' \in V'$, whence $w \equiv w' \bmod \mathrm{Orb}(T; v_1)$. Applying Fact 2.2 recursively to V' yields a decomposition of V as follows (see Gantmacher (1990), §7.4).

FACT 2.3 (modular cyclic decomposition). *For* $i \geq 1$ *let* $w_i \in V$ *and define* $V_i = \mathrm{Orb}(T; w_1) + \cdots + \mathrm{Orb}(T; w_i)$ *and* $T_i: V/V_i \to V/V_i$ *as* T *reduced modulo* V_i *(with* $V_0 = \{0\}$ *and* $T_0 = T$*). Assume that* k *is the smallest integer such that* $V_k = V$, *and for* $1 \leq i \leq k$ *we have* $\min(T_{i-1}; w_i + V_{i-1}) = \min(T_{i-1}; V/V_{i-1})$. *Then* $\min(T_{i-1}; w_i + V_{i-1}) = f_i$, *the* i*th invariant factor of* T *on* V, *for* $1 \leq i \leq k$.

If $w_1, \ldots, w_k$ are as in the above theorem, then we say that they generate a *modular cyclic basis*

$$w_1, Tw_1, \ldots, T^{d_1-1}w_1, w_2, Tw_2, \ldots, T^{d_2-1}w_2, \ldots, w_k, Tw_k, \ldots, T^{d_k-1}w_k$$

for V. Of course the summation $V = \mathrm{Orb}(T; w_1) + \cdots + \mathrm{Orb}(T; w_k)$ in Fact 2.3 is not direct, and once we have found $w_1, \ldots, w_k$, we must somehow "purify" them to get a direct sum. This is accomplished in §4.

Our approach to finding $w_1, \ldots, w_k \in V$ is to choose them randomly. This will only work when $\#\mathsf{K} \geq n^2$. When $\#\mathsf{K} < n^2$, we choose a small extension field F of K such that $\#\mathsf{F} \geq n^2$. This will be discussed in §5. For the remainder of this section, assume that $\#\mathsf{K} \geq n^2$.

LEMMA 2.4. *Let* W *be any vector space of dimension at most* n *over* K, *spanned by* $u_1, \ldots, u_n \in W$, *and let* $T: W \to W$ *be a* K*-linear map. Let* L *be a subset of* K *containing at least* n^2 *elements. Then*

$$\operatorname*{Prob}_{(a_1,\ldots,a_n)\in L^n} \left\{ \min\left(T; \sum_{1\leq i\leq n} a_i u_i\right) = \min(T; W) \right\} \geq 1 - \frac{1}{n}.$$

Proof. Let $y_1, \ldots, y_l \in W$ be such that

$$W = \mathrm{Orb}(T; y_1) \oplus \mathrm{Orb}(T; y_2) \oplus \cdots \oplus \mathrm{Orb}(T; y_l)$$

is the cyclic decomposition of W with respect to T, where T has l invariant factors. For $1 \leq i \leq l$, let $g_i = \min(T; y_i) \in \mathsf{K}[x]$ be the ith invariant factor of T, so $g_i \mid g_{i-1}$ for $2 \leq i \leq l$.

We choose random elements $a_1, \ldots, a_n \in L$ and assign $v = \sum_{1\leq i\leq n} a_i u_i \in W$. In what follows we only consider the component of v in $\mathrm{Orb}(T; y_1)$. Let $h_i \in \mathsf{K}[x]$ be such that the component of u_i in $\mathrm{Orb}(T; y_1)$ is $h_i \circ y_1$, with $\deg h_i < \deg g_1$, for $1 \leq i \leq l$. The component of v in $\mathrm{Orb}(T; y_1)$ is then $v_1 = \sum_{1\leq i\leq n} a_i h_i \circ y_1$. Let $T_1: \mathrm{Orb}(T; y_1) \to \mathrm{Orb}(T; y_1)$ be the restriction of T to $\mathrm{Orb}(T; y_1)$. Certainly $\min(T; v) = g_1$ when $\min(T_1; v_1) = g_1$; $\min(T; v)$ must annihilate *all* of v's components. When is $\min(T_1; v_1) = g_1$? Let $h = \sum_{1\leq i\leq n} a_i h_i$, so that $v_1 = h \circ y_1$. Clearly $g_1 \circ v_1 = (g_1 h) \circ y_1 = 0$ so $g = \min(T; v_1)$ divides g_1. Also, $(gh) \circ y_1 = 0$, which is true only when $gh \equiv 0 \bmod g_1$. If $\gcd(h, g_1) = 1$ then $g \equiv 0 \bmod g_1$, whence $g = g_1$. Conversely, if $\gcd(h, g_1) \neq 1$ then g is a proper divisor of g_1. In summary, $\min(T; v_1) = g_1$ if and only if $\gcd(h, g_1) = 1$.

To determine the probability that, for randomly chosen $a_1, \ldots, a_n \in L$, we have $\gcd(\sum_{1\leq i\leq n} a_i h_i, g_1) = 1$, consider $\rho = \sum_{1\leq i\leq n} x_i h_i \in \mathsf{K}[x_1, \ldots, x_n][x]$, in the indeter-

minates $x_1, \ldots, x_n, x$. To prove the theorem it is sufficient to show that

$$\operatorname*{Prob}_{(a_1,\ldots,a_n)\in L^n}\left\{\gcd(\rho(a_1,\ldots,a_n)(x), g_1(x)) = 1\right\} \geq 1 - \frac{1}{n}.$$

This is accomplished with the aid of resultants (see van der Waerden (1970), §5.8). The resultant of ρ and g_1, considered as polynomials in x with coefficients in the integral domain $\mathsf{K}[x_1, \ldots, x_n]$ is a polynomial $R \in \mathsf{K}[x_1, \ldots, x_n]$ of degree at most n. Moreover, $\gcd(\rho(a_1, \ldots, a_n)(x), g_1(x)) = 1$ if and only if $R(a_1, \ldots, a_n) \neq 0$.

The polynomial R is nonzero as follows. The component of y_1 in $\mathrm{Orb}(T; y_1)$ is y_1 itself, and hence has minimal polynomial g_1. Assume $y_1 = \sum_{1\leq j\leq n} b_j u_j$ for some $b_1, \ldots, b_n \in \mathsf{K}$. Then $\sum_{1\leq j\leq n} b_j h_j \equiv 1 \bmod g_1$, and since each h_j has degree less than g_1 by definition, $\sum_{1\leq j\leq n} b_j h_j = 1$. Thus $R(b_1, \ldots, b_n) \neq 0$.

We now apply Corollary 1 of Schwartz (1980) to obtain

$$\operatorname*{Prob}_{(a_1,\ldots,a_n)\in L^n}\left\{R(a_1, \ldots, a_n) \neq 0\right\} \geq 1 - \frac{n}{\#L} \geq 1 - \frac{1}{n}. \qquad \square$$

A modular cyclic decomposition is now constructed by simply choosing the generating vectors "randomly" from V. Actually, since we do not know the number k of invariant factors beforehand, we choose $w_1, \ldots, w_n \in V$ and consider the probability that the first k of these vectors generate V in the desired manner.

THEOREM 2.5. *Let L be a subset of K containing at least n^2 elements. For $1 \leq i \leq n$, randomly choose $w_i \in L^{n\times 1} \subseteq V$ and define $V_i = \mathrm{Orb}(T; w_1) + \cdots + \mathrm{Orb}(T; w_i)$ and $T_i\colon V/V_i \to V/V_i$ as T reduced modulo V_i (with $V_0 = \{0\}$ and $T_0 = T$). With probability at least $1/4$, for all $1 \leq i \leq k$ (where k is the number of invariant factors of T) the ith invariant factor f_i of T satisfies $f_i = \min(T_{i-1}; w_i + V_{i-1}) = \min(T_{i-1}; V/V_{i-1})$.*

Proof. First we note, for $1 \leq i \leq n$, that V_i is a T-invariant subspace, so V/V_i is well defined as a $\mathsf{K}[T]$-module. We show that for each $1 \leq i \leq n$ the probability that $\min(T_{i-1}; w_i + V_{i-1}) = \min(T_{i-1}; V/V_{i-1})$ is at least $1 - 1/n$. To see this, consider the standard basis $e_1, \ldots, e_n \in \mathsf{K}^{n\times 1}$ for V, i.e., e_i is the vector with a one in the ith row and zeros in all other rows. We have chosen $w_i = \sum_{1\leq i\leq n} a_i e_i$ for some randomly selected $a_1, \ldots, a_n \in L$. The vectors $e_1 + V_{i-1}, \ldots, e_n + V_{i-1}$ span V/V_{i-1} and

$$w_i + V_{i-1} = \sum_{1\leq i\leq n} a_i e_i + V_{i-1} = \sum_{1\leq i\leq n} a_i(e_i + V_{i-1}) \in V/V_{i-1}.$$

Thus, by Lemma 2.4,

$$\mathrm{Prob}\left\{\min(T_{i-1}; w_i + V_{i-1}) = \min(T_{i-1}; V/V_{i-1})\right\} \geq 1 - 1/n.$$

The number of components k in the cyclic decomposition is certainly at most n, whence

$$\prod_{1\leq i\leq k} \operatorname*{Prob}_{w_i\in L^{n\times 1}}\left\{\min(T_{i-1}; w_i + V_{i-1}) = \min(T_{i-1}; V/V_{i-1})\right\} \geq \left(1 - \frac{1}{n}\right)^n \geq \frac{1}{4}.$$

If all the w_i are chosen correctly then we also know, from Fact 2.3, that $f_i = \min(T_{i-1}; w_i + V_{i-1})$ for $1 \leq i \leq k$. $\square$

3. Computing the invariant factors of T on V. Assume now that we are given vectors $w_1, \ldots, w_n \in V$ for V such that $w_1, \ldots, w_k$ generate a modular cyclic basis for V, and let

V_i, T_i be as in Fact 2.3 for $0 \le i \le k$. How do we find k and the invariant factors $f_1, \ldots, f_k$? We adapt an algorithm of Keller-Gehrig (1985) to accomplish this with $O(\mathrm{MM}(n) \log n)$ operations in K.

FACT 3.1 (Keller-Gehrig (1985)). *Let $T \in \mathsf{K}^{n \times n}$ and $u_1, \ldots, u_n \in V$. Matrices $H_i \in \mathsf{K}^{n \times d_i}$ for $1 \le i \le n$ with the following properties can be computed with $O(\mathrm{MM}(n) \log n)$ operations in K:*

(i) $\sum_{1 \le i \le n} d_i = n$, *where $d_i \in \mathbb{N}$ may equal 0 for notational convenience;*

(ii) *for $1 \le i \le n$, $H_i = [u_i | T u_i | \cdots | T^{d_i - 1} u_i]$ if $d_i > 0$;*

(iii) *the columns of $H_1, \ldots, H_i$ form a basis of the $\mathsf{K}[x]$-module generated by $u_1, \ldots, u_i$, for $1 \le i \le n$ (that is, for $\mathrm{Orb}(T; u_1) + \cdots + \mathrm{Orb}(T; u_i)$).*

Keller-Gehrig's (1985) algorithm is essentially an asymptotically fast version of the algorithm of Danilevsky (1937) for computing the characteristic polynomial of a matrix (see Faddeev and Faddeeva (1963)). Applying Fact 3.1 to the vectors $w_1, \ldots, w_n$ yields k, bases for $V_1, \ldots, V_k$, and the degrees of the invariant factors $f_1, \ldots, f_k$. That is, the columns of $H_1, \ldots, H_i$ form a basis for $V_i = \mathrm{Orb}(T; w_1) + \cdots + \mathrm{Orb}(T; w_i)$ and $\deg f_i = d_i$ for $1 \le i \le k$. We know $d_i = 0$ for $k < i \le n$ since $w_1, \ldots, w_k$ are assumed to generate V as a $\mathsf{K}[x]$-module.

Let $H = [H_1 | H_2 | \cdots | H_k] \in \mathsf{K}^{n \times n}$, which we call a *modular cyclic transition matrix* for T. Combining Fact 3.1 with Theorem 2.5 gives a probabilistic algorithm to find a modular cyclic transition matrix, which is correct with probability at least $1/4$. We summarize this algorithm below.

Algorithm: `FindModCyc`

Input: $T \in \mathsf{K}^{n \times n}$, where $\#\mathsf{K} \ge n^2$ and L is a subset of K with $\#L \ge n^2$;

Output: a modular cyclic transition matrix $H = [H_1 | \cdots | H_k] \in \mathsf{K}^{n \times n}$ for T,

where $H_i = [w_i | \cdots | T^{d_i - 1} w_i] \in \mathsf{K}^{n \times d_i}$ for some $w_i \in V$ $(1 \le i \le k)$;

(1) Choose $w_1, \ldots, w_n$ randomly from $L^{n \times 1}$.

(2) Find $H = [H_1 | \cdots | H_k]$ such that $H_i = [w_i | \cdots | T^{d_i - 1} w_i] \in \mathsf{K}^{n \times d_i}$, where

- $\sum_{1 \le i \le k} d_i = n$
- for $1 \le i \le k$, the columns of $H_1, \ldots, H_i$ form a basis of the $\mathsf{K}[x]$-module generated by $w_1, \ldots, w_i$,

using Keller-Gehrig's algorithm.

End.

THEOREM 3.2. *Let $\#\mathsf{K} \ge n^2$. Given $T \in \mathsf{K}^{n \times n}$, the algorithm* `FindModCyc` *returns $w_1, \ldots, w_k \in V$ and matrices $H_i \in \mathsf{K}^{n \times d_i}$ for $1 \le i \le k$ defined as follows:*

(i) $\sum_{1 \le i \le k} d_i = n$ *with $d_i > 0$ for $1 \le i \le k$;*

(ii) $H_i = [w_i | \cdots | T^{d_i - 1} w_i]$ *for $1 \le i \le k$;*

(iii) *the columns of $H_1, \ldots, H_i$ form a basis for $V_i = \mathrm{Orb}(T; w_1) + \cdots + \mathrm{Orb}(T; w_i)$;*

(iv) $f_i = \min(T \bmod V_{i-1}; w_i + V_{i-1}) = \min(T \bmod V_{i-1}; V / V_{i-1})$ *for $1 \le i \le k$;*

i.e., H is a modular cyclic transition matrix for T. The algorithm is probabilistic, and returns the correct answer with probability at least $1/4$ (it may produce an incorrect answer). It requires $O(\mathrm{MM}(n) \log n)$ operations in K.

When H is a modular cyclic transition matrix it determines a change of basis on V under which T has the form

$$G = H^{-1}TH = \begin{pmatrix} C_{f_1} & B_2 & \cdots & B_k \\ & C_{f_2} & & \\ & & \ddots & \\ \multicolumn{2}{c}{0} & & C_{f_k} \end{pmatrix} \in \mathsf{K}^{n\times n}, \tag{3.1}$$

where $C_{f_i} \in \mathsf{K}^{d_i \times d_i}$ is the companion matrix of f_i, the ith invariant factor of T for $1 \le i \le k$ (see Keller-Gehrig (1985), §5). Each matrix $B_i \in \mathsf{K}^{t_i \times d_i}$, where $t_i = \sum_{1\le j<i} d_j$, is zero except for its last column; suppose this last column is

$$\vec{b}_i = (b_{i,1,0}, \ldots, b_{i,1,d_1-1}, b_{i,2,0}, \ldots, b_{i,2,d_2-1}, \ldots, b_{i,i-1,0}, \ldots, b_{i,i-1,d_{i-1}-1})^t \in \mathsf{K}^{t_i\times 1},$$

where $b_{ijl} \in \mathsf{K}$ for $2 \le i \le k$, $1 \le j < i$, and $0 \le l < d_j$. Under the change of basis induced by H, the vector $e_{t_i+j+1} \in \mathsf{K}^{n\times 1}$ (the column vector of all zeros except for one in the $(t_i + j + 1)$st row) in the basis given by the columns of H is the image of $T^j w_i$ in the standard basis for V.

The column $\vec{b}_i$ in B_i and the last column $(-a_{i,0}, \ldots, -a_{i,d_i-1})^t \in \mathsf{K}^{d_i\times 1}$ of C_{f_i} give the dependency of $T^{d_i} w_i$ on $T^l w_j$ for $1 \le j \le i$ and $0 \le l < d_i$ by

$$Ge_{t_i+d_i} = H^{-1}T^{d_i}w_i = H^{-1}\sum_{0\le l<d_i}(-a_{il})T^l w_i + H^{-1}\sum_{1\le j<i}\sum_{0\le l<d_j} c_{ijl}T^l w_j,$$

whence

$$T^{d_i}w_i + \sum_{0\le l<d_i} a_l T^l w_i = \sum_{1\le j<i}\sum_{0\le l<d_j} c_{ijl}T^l w_j,$$

or more simply

$$f_i \circ w_i = \sum_{1\le j<i} g_{ij} \circ w_j,$$

where $f_i = \sum_{0\le l\le d_i} a_{il}x^l$ and $g_{ij} = \sum_{0\le l<d_j} c_{ijl}x^l$ with $a_{il}, c_{ijl} \in \mathsf{K}$ for $1 \le j < i$ and $1 \le i \le k$. Thus $f_i \circ w_i \in V_{i-1}$, and since it has minimal degree, it is the ith invariant factor of T on V. Note that by construction, $\sum_{1\le j<i} \deg g_{ij} \le n$ for all $1 \le i \le k$, a fact which will be required in the next section.

THEOREM 3.3. *Let $w_1, \ldots, w_n \in V$ be a basis for V such that $w_1, \ldots, w_k$ generate a modular cyclic basis for V with $f_i = \min(T_{i-1}; w_i + V_{i-1})$ being the ith invariant factor of T for $1 \le i \le k$ and V_i, T_i as in Fact 2.3. Suppose also that we have computed a modular cyclic transition matrix for T. Then we can determine $k \in \mathbb{N}$, $f_1, \ldots, f_k \in \mathsf{K}[x]$, and $g_{ij} \in \mathsf{K}[x]$ for $1 \le j < i \le k$ such that $f_i \circ w_i = \sum_{1\le j<i} g_{ij} \circ w_j$ with $O(\mathrm{MM}(n))$ operations in K. Furthermore, $\sum_{1\le j<i} \deg g_{ij} \le n$ for all $1 \le i \le k$.*

Combining this theorem with Theorem 3.2 gives a Monte-Carlo-type probabilistic algorithm for computing the Frobenius form of T. Simply use the algorithm `FindModCyc` to find a modular cyclic transition matrix $H \in \mathsf{K}^{n\times n}$ for T. With probability at least 1/4

the companion blocks of the the invariant factors $f_1, \ldots, f_k \in \mathsf{K}[x]$ of T are along the diagonal of $H^{-1}TH$, from which we can construct the Frobenius form S of T. Of course, with positive probability we will get an erroneous Frobenius form, and since we do not get an invertible matrix $U \in \mathsf{K}^{n\times n}$ such that $S = U^{-1}TU$ from this procedure, we have no way of verifying that S is correct. This problem is addressed in the next section.

4. Purifying the modular decomposition. Once again, let $w_1, \ldots, w_k \in V$ generate a modular cyclic basis for V, where V_i and T_i are as in Fact 2.3 for $0 \le i \le n$, and $f_i = \min(T_{i-1}; w_i + V_{i-1})$ is the ith invariant factor of T on V for $1 \le i \le k$. Furthermore, assume that $f_i \circ w_i = \sum_{1\le j<i} g_{ij} \circ w_j$ for some $g_{ij} \in \mathsf{K}[x]$ for all $1 \le j < i \le k$. This information can all be computed with the algorithm of the previous section (see Theorem 3.3), along with the matrix H defined there, whose columns give a modular decomposition basis for V. In this section vectors $v_1, \ldots, v_k \in V$ are constructed such that

$$V = \mathrm{Orb}(T; v_1) \oplus \mathrm{Orb}(T; v_2) \oplus \cdots \oplus \mathrm{Orb}(T; v_k),$$

and $f_i = \min(T; v_i)$ for $1 \le i \le k$. The key fact required is from Hoffman and Kunze (1971), §7.2, Theorem 3, Step 2.

FACT 4.1. *If $w_i \in V$, and $f_i, g_{ij} \in \mathsf{K}[x]$ are as above, then $f_i \mid g_{ij}$ for $1 \le j < i \le k$.*

Applying Fact 4.1, assume that $g_{ij} = f_i h_{ij}$ for some $h_{ij} \in \mathsf{K}[x]$ for all $1 \le j < i \le k$. We claim that the vectors $v_i = w_i - \sum_{1\le j<i} h_{ij} \circ w_j \in V$ for $1 \le i \le k$ are the ones desired.

THEOREM 4.2. *Let $w_1, \ldots, w_k \in V$ and $V_i = \mathrm{Orb}(T; w_1) + \cdots + \mathrm{Orb}(T; w_i)$ such that $V_k = V$ and $V_{k-1} \ne V$. Also, let $f_i = \min(T_{i-1}; w_i + V_{i-1})$ such that $f_i \circ w_i = \sum_{1\le j<i} g_{ij} \circ w_j$ and $g_{ij} = f_i h_{ij}$ for $g_{ij}, h_{ij} \in \mathsf{K}[x]$ with $1 \le j < i \le k$. If $v_i = w_i - \sum_{0\le j<i} h_{ij} \circ w_j$, then for $1 \le i \le k$,*

(i) $f_i = \min(T; v_i)$,

(ii) $V_i = \mathrm{Orb}(T; v_1) \oplus \mathrm{Orb}(T; v_2) \oplus \cdots \oplus \mathrm{Orb}(T; v_i)$.

Proof. To show (i), note that for any i with $1 \le i \le k$ we have

$$f_i \circ v_i = f_i \circ w_i - f_i \circ \left(\sum_{1\le j<i} h_{ij} \circ w_j \right) = f_i \circ w_i - \sum_{1\le j<i} f_i h_{ij} \circ w_j$$
$$= f_i \circ w_i - \sum_{1\le j<i} g_{ij} \circ w_j = 0.$$

Furthermore, since f_i is the polynomial of least degree such that $f_i \circ w_i \in V_{i-1}$, it follows that $f_i = \min(T; v_i)$.

To show (ii), we must show that $V_{i-1} \cap \mathrm{Orb}(T; v_i) = \{0\}$, or equivalently that if $g \circ v_i \in V_{i-1}$ then $g \circ v_i = 0$ for any $g \in \mathsf{K}[x]$. Suppose $g \circ v_i \in V_{i-1}$ for some polynomial $g \in \mathsf{K}[x]$. This is true only if $g \circ w_i \in V_{i-1}$, since $v_i \equiv w_i \bmod V_{i-1}$. But $f_i = \min(T_{i-1}; w_i + V_{i-1})$, so $f_i \mid g$, and $g \circ v_i = 0$. Then $V_{i-1} \cap \mathrm{Orb}(T; v_i) = \{0\}$ and $V_i = V_{i-1} \oplus \mathrm{Orb}(T; v_i)$. □

Note that this theorem does not assume that $w_1, \ldots, w_k$ generate a modular cyclic basis for V, only that they generate V as a $\mathsf{K}[x]$-module.

To compute $v_1, \ldots, v_k$, first compute $h_{ij} = g_{ij}/f_i \in \mathsf{K}[x]$ for all $1 \le j < i \le k$. For each $1 \le i \le k$ we note that $\sum_{1\le j<i} \deg g_{ij} \le n$, so we can compute all g_{ij} for $1 \le j < i$ with $O(\mathrm{M}(n))$ operations in K. The number k of invariant factors is at most n, so all the g_{ij}'s can be found with $O(n\mathrm{M}(n))$ operations in K.

Computing the h_{ij}'s gives a representation of the v_i's in the modular decomposition basis for V given by the columns of H, not in the original (standard) basis for V. To find v_i in the standard basis, recall

$$v_i = w_i - \sum_{1\le j<i} h_{ij} \circ w_j = w_i - \sum_{1\le j<i} \sum_{0\le l<d_j} c_{ijl} T^l w_j.$$

We compute v_i by the matrix-vector product $v_i = H\bar{v}_i \in V$, where

$$\bar{v}_i = (-c_{i,1,1}, \ldots, -c_{i,1,d_2-1}, \ldots, -c_{i,i-1,1}, \ldots, -c_{i,i-1,d_{i-1}}, 1, 0, \ldots, 0)^t \in \mathsf{K}^{n\times 1}.$$

All of $v_1, \ldots, v_k$ can now be computed by a single matrix product

$$H \cdot [\bar{v}_1 | \ldots | \bar{v}_k] = [v_1 | \ldots | v_k]$$

with $O(\mathrm{MM}(n))$ operations in K.

THEOREM 4.3. *Given $w_i \in V$ and $f_i, g_{ij} \in \mathsf{K}[x]$ for $1 \le i \le k$ and $1 \le j < i$ as above, we can compute $v_1, \ldots, v_k \in V$ such that*

$$V = \mathrm{Orb}(T; v_1) \oplus \cdots \oplus \mathrm{Orb}(T; v_k)$$

and $f_i = \min(T; v_i)$ with $O(\mathrm{MM}(n) + n\mathrm{M}(n))$ operations in K.

5. Computing the Frobenius form. The complete algorithm for computing the Frobenius normal form of any $T \in \mathsf{K}^{n\times n}$, where $\#\mathsf{K} \ge n^2$, can now be stated. A modification which works over smaller fields is presented in what follows.

Algorithm: `FrobeniusForm`

Input: $T \in \mathsf{K}^{n\times n}$, where $\#\mathsf{K} \ge n^2$;

Output: – The Frobenius form $S \in \mathsf{K}^{n\times n}$ of T,
– an invertible $U \in \mathsf{K}^{n\times n}$ such that $S = U^{-1}TU$;

(1) Using `FindModCyc`, compute a modular cyclic transition matrix $H = [H_1 | \cdots | H_k] \in \mathsf{K}^{n\times n}$, where
- $H_i = \left[w_i | Tw_i | \cdots | T^{d_i-1}w_i\right] \in \mathsf{K}^{n\times d_i}$ for $1 \le i \le k$,
- the columns of $H_1, \ldots, H_i$ span the $\mathsf{K}[x]$-module generated by the vectors $w_1, \ldots, w_i$ for $1 \le i \le k$;

An erroneous computation of H will be detected in step (2) or (4).

(2) If H is not invertible then quit, returning "failure";

(3) Compute $f_i, g_{ij} \in \mathsf{K}[x]$ from $G = H^{-1}TH$ such that $f_1 \circ w_1 = 0$ and $f_i \circ w_i = \sum_{1\le j<i} g_{ij} \circ w_j$ for $2 \le i \le k$ and $1 \le j < i$;

(4) If $f_i \nmid f_{i-1}$ for some $2 \le i \le k$, or $f_i \nmid g_{ij}$ for some $1 \le j < i$ and $1 \le i \le k$, then return "failure" and quit;

(5) Compute $h_{ij} = g_{ij}/f_i$ for $1 \le j < i$ and $1 \le i \le k$;

(6) Compute $v_i = w_i - \sum_{1\le j<i} h_{ij} \circ w_j$ for $1 \le i \le k$;

(7) Compute the matrix $U \in \mathsf{K}^{n\times n}$ as in 2.1;

(8) Output the Frobenius form $S = \mathrm{diag}(C_{f_1}, \ldots, C_{f_k})$ and the transition matrix U (where C_{f_i} is the companion matrix of f_i for $1 \le i \le k$).

End.

THEOREM 5.1. *Let $T \in \mathsf{K}^{n\times n}$ over any field K with at least n^2 elements. With probability at least $1/4$,* `FrobeniusForm` *returns the Frobenius form S of T and an invertible $U \in \mathsf{K}^{n\times n}$ such that $S = U^{-1}TU$. Otherwise the algorithm reports "failure." In either case $O(\mathrm{MM}(n)\log n + n\mathrm{M}(n))$ operations in K are sufficient.*

Proof. First we prove the correctness of the output. We employ the algorithm `FindModCyc`, analyzed in Theorem 3.2, to compute a modular cyclic transition matrix

H, which will be correct with probability at least $1/4$. Assume for now that H is computed correctly. By Fact 4.1 we know $f_i \mid g_{ij}$ for $1 \le j < i$ and $1 \le i \le k$, and certainly $f_i \mid f_{i-1}$ for $2 \le i \le k$. Theorem 4.2 guarantees that $v_1, \ldots, v_k$ generate the cyclic composition factors of V, and Gantmacher (1990), §7.5.2, shows that $S = U^{-1}TU$ is in Frobenius normal form.

If H is incorrectly computed, the algorithm reports failure in step (2) or (4). Certainly if H is singular then it is incorrect, and this is detected in step (2). Otherwise, assume that the tests in step (4) are passed, but $w_1, \ldots, w_k$ do not generate a modular cyclic basis. By Theorem 4.2, $V = \mathrm{Orb}(T; v_1) \oplus \cdots \oplus \mathrm{Orb}(T; v_k)$. Let $l \ge 1$ be the smallest integer such that $f_l \ne \bar{f}_l$, where $\bar{f}_l$ is the the actual lth invariant factor of T on V. In the vector space

$$\bar{V}_l = \mathrm{Orb}(T; v_l) \oplus \cdots \oplus \mathrm{Orb}(T; v_k) \cong V/V_{l-1}$$

there exists a vector $w \in \bar{V}$ with $\min(T_{l-1}; w + V_{l-1}) = \bar{f}_l$ (where $T_{l-1}\colon V/V_{l-1} \to V/V_{l-1}$ is T reduced modulo V_{l-1}). Since $f_j \mid f_l$ for $l \le j \le k$, we know $f_l = \min(T_{l-1}; \bar{V}_l) = \bar{f}_l$, a contradiction.

By Theorem 3.2, step (1) requires $O(\mathrm{MM}(n) \log n)$ operations in K. Using Theorem 3.3, steps (2)–(4) require $O(\mathrm{MM}(n))$ operations in K (H is singular exactly when its last column is zero). By Theorem 4.3 steps (5) and (6) require $O(\mathrm{MM}(n))$ operations in K to complete. The matrix U in step (7) can be found using Fact 3.1, with $O(\mathrm{MM}(n) \log n)$ operations in K. □

The algorithm `FrobeniusForm` requires that the field K contain at least n^2 elements. If this is not the case, and $q = \#\mathsf{K} < n^2$, the chances of choosing $w_1, \ldots, w_k$ correctly may be very low. To remedy this we construct a field extension F of K containing n^2 elements. We then run the algorithm `FrobeniusForm` on $T \in \mathsf{F}^{n\times n}$. By Theorem 5.1 this correctly returns the Frobenius form $S \in \mathsf{F}^{n\times n}$ and an invertible $U \in \mathsf{F}^{n\times n}$ such that $S = U^{-1}TU$ with probability at least $1/4$. However, the Frobenius form of T is a unique rational invariant of T, regardless of the field in which the elements of T are embedded. Thus $S \in \mathsf{K}^{n\times n}$ and this is precisely what we are looking for. The only drawback to this technique, aside from the slightly increased cost of operations in F, is that U is not necessarily a matrix over K.

To construct the field $\mathsf{F} \supseteq \mathsf{K}$, we use the deterministic algorithm of Shoup (1993) to find a polynomial $\psi \in \mathsf{K}[x]$ of degree $\lceil 2 \log_q n \rceil$. Shoup's algorithm requires $O\tilde{}(n)$ operations in K. Let $\mathsf{F} = \mathsf{K}[x]/(\psi)$, a finite field with at least n^2 elements, where each element is represented by a polynomial of degree less than $\lceil 2 \log_q n \rceil$. Elements of K are simply represented as constant polynomials, giving a trivial embedding of K into F. Additions and subtractions in F require $O(\log_q n)$ operations in K, while multiplications in F require $O(\mathrm{M}(\log_q n))$ operations in K and divisions in F require $O(\mathrm{M}(\log_q n) \log\log_q n)$ operations in K.

THEOREM 5.2. *Let $T \in \mathsf{K}^{n\times n}$, where $q = \#\mathsf{K} < n^2$. The modification of* `FrobeniusForm` *described above computes the Frobenius form $S \in \mathsf{K}^{n\times n}$ of T, and an invertible matrix $U \in \mathsf{F}^{n\times n}$ such that $S = U^{-1}TU$. The field $\mathsf{F} = \mathsf{K}[x]/(\psi)$ is an extension of K, given by an irreducible monic $\psi \in \mathsf{K}[x]$ of degree $\lceil 2 \log_q n \rceil$. The algorithm succeeds with probability at least $1/4$, and otherwise reports "failure." It requires $O((\mathrm{MM}(n) \log n + n\mathrm{M}(n)) \cdot \mathrm{M}(\log_q n) \log\log_q n)$ or $O\tilde{}(\mathrm{MM}(n))$ operations in K.*

6. A processor-efficient parallel algorithm. In this section we exhibit a processor-efficient parallel algorithm for finding the Frobenius form of a matrix. Recall that computing the Frobenius form of a $T \in \mathsf{K}^{n\times n}$ is at least as hard as matrix multiplication, since from the Frobenius form we can quickly compute the determinant of T, which is known to be as hard as multiplying matrices (see Baur and Strassen (1982)). We demonstrate that our algorithm for computing the Frobenius form of any $T \in \mathsf{K}^{n\times n}$ can be implemented in a processor-efficient manner and can be executed in (Las Vegas) time $(\log n)^{O(1)}$ using $O(\mathrm{MM}(n))$ processors.

Actually, the only part of the algorithm `FrobeniusForm` which requires modification, other than specifying an implementation technique for each step from the "toolkit" of known parallel algorithms, is the subroutine `FindModCyc` used in step (1). The routine of Keller-Gehrig's (1985), used in step (2) of `FindModCyc` in §3, is the heart of his asymptotically fast version of Danilevsky's (1937) algorithm for finding the characteristic polynomial of a matrix. It is not immediately clear that Keller-Gehrig's algorithm can be implemented in a processor-efficient manner. In this section we exhibit fast parallel versions of Keller-Gehrig's algorithm for computing the characteristic polynomial of a matrix and of our routine `FindModCyc`, both of which are processor-efficient. We then give the implementation details of a processor-efficient version of `FrobeniusForm`.

We begin by collecting a number of useful parallel algorithms which we require as subroutines.

FACT 6.1. *Suppose* K *is a field with characteristic* p.

(i) *Suppose that there is a bilinear algorithm which, given* $A, B \in \mathsf{K}^{n\times n}$, *computes* AB *with* $O(\mathrm{MM}(n))$ *operations in* K *sequentially (for a definition of bilinear algorithm see Borodin and Munro* (1975)); *then there exists a processor-efficient parallel algorithm to compute* AB *on* $O(\mathrm{MM}(n)n^{\epsilon})$ *processors in time* $O(\log n)$ *for any* $\epsilon > 0$.

(ii) *Suppose* $\#\mathsf{K} \geq n$ *and* $A \in \mathsf{K}^{n\times n}$ *is invertible. If* $p \geq n$ *or* $p = 0$ *then we can compute* A^{-1} *with* $O(\mathrm{MM}(n))$ *processors in time* $O(\log^2 n)$. *If* $2 \leq p \leq n$, *there is a Las-Vegas-type probabilistic algorithm to compute* A^{-1} *with time* $O(\log^3 n/\log p)$ *on* $O(\mathrm{MM}(n))$ *processors.*

(iii) *If* $\#\mathsf{K} \geq n$, *then given vectors* $v_1, \ldots, v_m \in \mathsf{K}^{n\times 1}$, *where* $m \leq 2n$, *we can find the lexicographically first subset of* $v_1, \ldots, v_m$, *which form a basis for the vector space spanned by* $v_1, \ldots, v_m$. *If* $p \geq n$ *or* $p = 0$, *this can be accomplished with* $O(\log^4 n)$ *time on* $O(\mathrm{MM}(n)/\log n)$ *processors. If* $2 \leq p < n$, *this can be done with a Las Vegas algorithm requiring time* $O(\log^5 n/\log p)$ *on* $O(\mathrm{MM}(n)/\log n)$ *processors.*

(iv) *Given two polynomials* $f, g \in \mathsf{K}[x]$ *of degree at most* n, *we can compute* fg *in time* $O(\log n)$ *with* $O(n \log\log n)$ *processors.*

(v) *Given two polynomials* $f, g \in \mathsf{K}[x]$ *of degree at most* n, *we can compute the unique* $Q, R \in \mathsf{K}[x]$ *such that* $\deg R < \deg g$ *and* $f = Qg + R$ *in time* $O(\log n)$ *on* $O(n \log n)$ *processors.*

Proof. Part (i) is shown by Pan and Reif (1985), Theorem A.1. A processor-efficient algorithm for matrix inversion is given by Kaltofen and Pan (1991) and Kaltofen and Pan (1992). Eberly (1991) shows (iii). The fast polynomial multiplication algorithms of Cantor and Kaltofen (1991) also have parallel time and processor bounds as stated in part (iv). Part (v) is shown by Bini and Pan (1992). □

We make the assumption, based on Fact 6.1(i), that there exists an algorithm which requires time $O(\log n)$ on $O(\mathrm{MM}(n))$ processors to multiply two $n \times n$ matrices over a field K.

Assume that we are given vectors $u_1, \ldots, u_n \in V$. As in Fact 3.1, we want to find matrices $H_i \in \mathsf{K}^{n\times d_i}$ for $1 \leq i \leq n$ with the following properties:

(i) $\sum_{1\leq i\leq n} d_i = n$, where $d_i \in \mathbb{N}$ may equal 0 for notational convenience;

(ii) for $1 \leq i \leq n$, $H_i = [u_i \mid Tu_i \mid \cdots \mid T^{d_i-1}u_i]$ if $d_i > 0$;

(iii) the columns of $H_1, \ldots, H_i$ form a basis of the $\mathsf{K}[x]$-module generated by $u_1, \ldots, u_i$ for $1 \leq i \leq n$.

We compute $H = [H_1 \mid \cdots \mid H_n]$ by computing matrices $H^{(0)}, H^{(1)}, \ldots, H^{(r)} \in \mathsf{K}^{n\times n}$ in sequence, where $r = \lceil \log_2 n \rceil$, $H^{(0)} = T$, and $H^{(r)} = H$. At step 0 we have $H^{(0)} =$

$[H_1^{(0)} \mid \cdots \mid H_n^{(0)}]$, where $H_i^{(0)}$ is the ith column of T for $1 \le i \le n$. The idea is that at the end of stage j we will have computed $H^{(j)} = [H_1^{(j)} \mid \cdots \mid H_n^{(j)}]$, where

$$H_i^{(j)} = \left[u_i \mid Tu_i \mid \cdots \mid T^{d_i^{(j)}-1}u_i\right] \in \mathsf{K}^{n \times d_i^{(j)}} \quad \text{for } 1 \le i \le n.$$

Here $d_i^{(j)}$ is the smallest integer less than 2^j such that $T^{d_i^{(j)}}u_i$ lies in the vector space spanned by $u_i, Tu_i, \ldots, T^{d_i^{(j)}-1}u_i$ along with all the columns of $H_1^{(j)}, \ldots, H_{i-1}^{(j)}$. If $u_i, Tu_i, \ldots, T^{d_i^{(j)}}u_i$ and the columns of $H_1^{(j)}, \ldots, H_{i-1}^{(j)}$ are linearly independent then $d_i^{(j)} = 2^j$. This can be expressed more concisely in the language of modular vector spaces. Let $V_{i-1}^{(j)}$ be the vector space spanned by the columns of $H_1^{(j)}, \ldots, H_{i-1}^{(j)}$. Then $d_i^{(j)}$ is the smaller of 2^j and $\dim \mathrm{Orb}(T \bmod V_{i-1}^{(j)}; u_i \bmod V_{i-1}^{(j)})$. Recall that we allow $d_i^{(j)}$ to be zero, meaning that u_i is linearly dependent on the columns of $H_1^{(j)}, \ldots, H_{i-1}^{(j)}$. Since $2^r \ge n$, it is clear that $H^{(r)}$ will have the desired properties.

To find $H^{(j)}$ after computing $H^{(j-1)}$ (for $1 \le j \le r$), we first compute matrices $J_1^{(j)}, \ldots, J_n^{(j)}$ as follows:

$$J_i^{(j)} = \begin{cases} H_i^{(j-1)} & \text{if } H_i^{(j-1)} \text{ has fewer than } 2^j \text{ columns,} \\ [H_i^{(j-1)} \mid T^{2^j} H_i^{(j-1)}] & \text{otherwise.} \end{cases}$$

This doubles the number of iterates of u_i under T if this is required. Note that if $T^l u_i$ is linearly dependent upon V_i together with $u_i, Tu_i, \ldots, T^{l-1}u_i$ then so is $T^{l+1}u_i$. Thus, to find $d_1^{(j)}, \ldots, d_n^{(j)}$ we identify and mark the lexicographically first set of n linearly independent columns of

$$J^{(j)} = \left[J_1^{(j)} \mid \cdots \mid J_n^{(j)}\right].$$

The last column marked in $J_i^{(j)}$ is indexed by $d_i^{(j)}$. If none of the identified columns are in $J_i^{(j)}$ for some i then $d_i^{(j)} = 0$. Now let $H_i^{(j)} \in \mathsf{K}^{n \times d_i}$ consist of those columns of $J_i^{(j)}$ marked in this process.

This can be implemented in a processor-efficient manner using Fact 6.1. At stage j we compute $T^{2^j} = T^{2^{j-1}} \cdot T^{2^{j-1}}$; then we compute $J^{(j)}$ as described above from the product $T^{2^j} H^{(j-1)}$. The lexicographically first subset of the columns of $J^{(j)}$ are identified with the algorithm of Eberly (1991) (Fact 6.1(iii) in this paper), and $H^{(j)} \in \mathsf{K}^{n \times n}$ is constructed as above. This is repeated $r = \lceil \log_2 n \rceil$ times.

THEOREM 6.2. *If $p \ge n$ or $p = 0$ we can compute $H \in \mathsf{K}^{n \times n}$ and $d_1, \ldots, d_n \in \mathbb{N}$ as in Fact 3.1 in time $O(\log^4 n)$ on $O(\mathrm{MM}(n))$ processors. If $2 \le p \le n$, this algorithm requires time $O(\log^5 n / \log p)$ on $O(\mathrm{MM}(n))$ processors.*

Combining Theorem 6.2 with Theorem 2.5 allows us to implement the algorithm `FindModCyc` in §3 in a processor-efficient manner.

THEOREM 6.3. *Let $\#\mathsf{K} \ge n^2$. Given $T \in \mathsf{K}^{n \times n}$, the algorithm* `FindModCyc`, *implemented in a processor-efficient manner as described above, returns $w_1, \ldots, w_k \in V$ and matrices $H_i \in \mathsf{K}^{n \times d_i}$ for $1 \le i \le k$ defined as follows:*

(i) $\sum_{1 \le i \le k} d_i = n$ *with* $d_i > 0$ *for* $1 \le i \le k$.

(ii) $H_i = [w_i | \cdots | T^{d_i - 1} w_i]$ *for* $1 \le i \le k$.

(iii) *The columns of $H_1, \ldots, H_i$ form a basis for $V_i = \mathrm{Orb}(T; w_1) + \cdots + \mathrm{Orb}(T; w_i)$.*

(iv) $f_i = \min(T \bmod V_{i-1}; w_i + V_{i-1}) = \min(T \bmod V_{i-1}; V/V_{i-1})$ *for* $1 \le i \le k$.

The algorithm is probabilistic and returns the correct answer with probability at least $1/4$ *(it may produce an incorrect answer). If* $p \geq n$ *or* $p = 0$, *it requires time* $O(\log^4 n)$ *on* $O(\mathrm{MM}(n))$ *processors. If* $2 \leq p \leq n$, *it requires time* $O(\log^5 n / \log p)$ *on* $O(\mathrm{MM}(n))$ *processors.*

We now examine the parallel cost of the algorithm `FrobeniusForm`.

THEOREM 6.4. *The Las Vegas algorithm* `FrobeniusForm` *can be implemented in parallel in a processor-efficient manner. Let* $T \in \mathsf{K}^{n \times n}$, *where* K *is a field with* $\#\mathsf{K} \geq n^2$.

(i) *If* $p \geq n$ *or* $p = 0$ *then the algorithm* `FrobeniusForm` *requires time* $O(\log^4 n)$ *on* $O(\mathrm{MM}(n))$ *processors.*

(ii) *If* $2 \leq p < n$ *then the algorithm* `FrobeniusForm` *requires time* $O(\log^5 n / \log p)$ *on* $O(\mathrm{MM}(n))$ *processors.*

Proof. The statement of the parallel algorithm does not vary from the description in §5. Only the manner in which each step is executed changes, and we address these changes now. Step (1) is accomplished using the processor-efficient implementation of `FindModCyc` described in Theorem 6.3. The matrix H is singular exactly when its last column is zero, and this can be tested with constant time and work to perform step (2). As in the sequential algorithm, the coefficients of f_i and g_{ij} for $2 \leq i \leq k$ and $1 \leq j < i$ are read from $G = H^{-1}TH$ as in Theorem 3.3, and G can be computed within the desired parallel time and processor bounds by Fact 6.1. Steps (4) and (5) can be done using processor-efficient polynomial division algorithms, as specified in Fact 6.1 parts (iv) and (v). Step (6) can be performed by a single matrix product as in Theorem 4.3, and so can be done in a processor-efficient manner. Applying Theorem 6.3 gives us U in step (7). □

When $\#\mathsf{K} < n^2$, similar theorems hold with running times and processor requirements multiplied by a small power of $\log_q n$.

THEOREM 6.5. *Let* K *be a field with* $q < n^2$ *and* $T \in \mathsf{K}^{n \times n}$. *We can implement the algorithm* `FrobeniusForm`, *as modified in Theorem* 5.2, *in a processor-efficient manner.*

(i) *If* $p \geq n$ *or* $p = 0$, *it requires time* $O(\log^4(n) \log\log_q n)$ *on* $O(\mathrm{MM}(n) \cdot \mathrm{M}(\log_q n))$ *processors.*

(ii) *If* $2 \leq p < n$, *it requires time* $O(\log^5(n) \log\log_q(n) / \log p)$ *on* $O(\mathrm{MM}(n) \cdot \mathrm{M}(\log_q n))$ *processors.*

Proof. The proof is the same as that of Theorem 6.4 except that we work in an extension F of K of algebraic degree $\lceil \log_q n \rceil$ over K, which does contain $O(n^2)$ elements (see Theorem 5.2). Each operation in F requires $O(\log\log_q n)$ operations in K on $O(\mathrm{M}(\log_q n))$ processors. □

Theorem 6.3 also has the following important corollary, essentially a processor-efficient version of the algorithms of Danilevsky (1937) and Keller-Gehrig (1985) for computing the characteristic polynomial of a matrix.

COROLLARY 6.6. *There is a processor-efficient Las Vegas algorithm for computing the characteristic polynomial* $f \in \mathsf{K}[x]$ *of any matrix* $T \in \mathsf{K}^{n \times n}$ *over any field* K. *If* $p \geq n$ *or* $p = 0$, *we can compute* f *in time* $O(\log^4 n)$ *on* $O(\mathrm{MM}(n))$ *processors. If* $2 \leq p < n$, *this algorithm requires time* $O(\log^5 n / \log p)$ *on* $O(\mathrm{MM}(n))$ *processors.*

Proof. We first find a matrix $H \in \mathsf{K}^{n \times n}$ as in Theorem 6.2 such that $G = H^{-1}TH$ is as in (3.1). The inverse of H and product $f_1 \ldots f_k$ can be computed within the desired time and processor bounds using the methods described in Fact 6.1. □

Note that the characteristic polynomial of $T \in \mathsf{K}^{n \times n}$ can also be computed as the product of the invariant factors of T, which can be read from the Frobenius form of T. The above corollary is a slight improvement on this for small fields. It relies on the fact that we do not need $H^{-1}TH$ to be in Frobenius form; the product of the polynomials whose companion matrices are on the diagonal of $H^{-1}TH$ always equals the characteristic polynomial of T.

7. Fast evaluation of matrix functions. We now consider applications of our algorithm for finding the Frobenius form to evaluating polynomials at matrices, determining matrix similarity, and finding the minimal polynomial of matrices. New algorithms requiring almost optimal time are presented. We also give lower bounds for the problems of evaluating a polynomial at matrix and finding the minimal polynomial of a matrix, which matches our upper bounds within polylogarithmic factors.

Evaluating polynomials at matrices. Computing the Frobenius form $S \in \mathsf{K}^{n\times n}$ of a matrix $T \in \mathsf{K}^{n\times n}$ allows quick evaluation of $g(T) \in \mathsf{K}^{n\times n}$ for any polynomial $g \in \mathsf{K}[x]$. The problem of evaluating polynomials at matrices is certainly not new. Paterson and Stockmeyer (1973) give an algorithm to evaluate a polynomial $g \in \mathsf{K}[x]$ at any point in a ring extension $\mathfrak{R}$ of K, which requires $O(\sqrt{r})$ nonscalar multiplications in $\mathfrak{R}$, where $r = \deg g$. They apply their algorithm to evaluate $g(T)$ at any $T \in \mathsf{K}^{n\times n}$ with $O(\mathrm{MM}(n)\sqrt{r})$ operations in K (see also Brent and Kung (1978)).

Theorem 7.2 gives a substantial improvement over this, especially when r is large, allowing evaluation of $g(T)$ with $O\tilde{}(\mathrm{MM}(n) + r)$ operations in K. The well-known observation required is that if $S = U^{-1}TU$ for some invertible $U \in \mathsf{K}^{n\times n}$, and $S = \mathrm{diag}(C_{f_1}, \ldots, C_{f_k})$ as in 1.1, then

$$g(T) = U \cdot g(S) \cdot U^{-1} = U \cdot \mathrm{diag}(g(C_{f_1}), \ldots, g(C_{f_k})) \cdot U^{-1}.$$

Furthermore, each $g(C_{f_i})$ can be evaluated quickly once $g \bmod f_i$ has been computed, and $g(S)$ can be computed with $O(n^2)$ operations in K when $\deg g \le n$.

LEMMA 7.1. *Let $C \in \mathsf{K}^{s\times s}$ be the companion matrix of $h \in \mathsf{K}[x]$, where $\deg h = s$. Also, let $g \in \mathsf{K}[x]$ have degree less than s. Then we can compute $g(C)$ with $O(s^2)$ operations in K or in parallel time $O(\log s)$ on $O(s\mathrm{M}(s))$ processors.*

Proof. The vector space $\mathsf{K}^{s\times 1}$ has a $\mathsf{K}[x]$-module structure, where any $f \in \mathsf{K}[x]$ acts on $\mathsf{K}^{s\times 1}$ as $f(C) \in \mathsf{K}^{s\times s}$. As a $\mathsf{K}[x]$-module, $\mathsf{K}^{s\times 1}$ is isomorphic to $\mathsf{K}[x]/(h)$—simply map $v = (b_0, \ldots, b_{s-1}) \in \mathsf{K}^{s\times 1}$ to $\varphi(v) = \sum_{0\le i<s} b_i x^i \bmod h$. It is easy to show that $x\varphi(v) \equiv \varphi(Cv) \bmod h$, and therefore that $f \cdot \varphi(v) = \varphi(f(C)v)$ for any $f \in \mathsf{K}[x]$. In particular, $\varphi(e_i) = x^{i-1}$, where $e_i \in \mathsf{K}^{n\times 1}$ is zero except for a one in the ith row, and hence $\varphi(g(C)e_i) \equiv g \cdot x^{i-1} \bmod h$ for $1 \le i \le s$. Since $g(C)e_i$ is the ith column of $g(C)$, we can find $g(C)$ by computing $g \cdot x^i \bmod h$ for $0 \le i < s$. This can be accomplished with $O(s^2)$ operations in K by realizing that computing $C \cdot v$ for any $v \in \mathsf{K}^{s\times 1}$, and hence multiplying by x modulo h, requires only $O(s)$ operations in K (C has only $2s$ nonzero entries). It can be done in time $O(\log s)$ on $O(s\mathrm{M}(s))$ processors using the polynomial multiplication algorithm of Cantor and Kaltofen (1991). □

THEOREM 7.2. *Given $g \in \mathsf{K}[x]$ of degree r and $T \in \mathsf{K}^{n\times n}$, we can compute $g(T) \in \mathsf{K}^{n\times n}$ with a Las-Vegas-type probabilistic algorithm requiring $O(\mathrm{MM}(n)\log n + n\mathrm{M}(n) + \mathrm{M}(r))$ operations in K. Here K is any field with at least n^2 elements.*

Proof. First compute the Frobenius form $S \in \mathsf{K}^{n\times n}$ of T and an invertible $U \in \mathsf{F}^{n\times n}$ such that $U^{-1}TU = S$. This requires $O(\mathrm{MM}(n)\log n + n\mathrm{M}(n))$ operations in K using Theorem 5.1. For each invariant factor $f_i \in \mathsf{K}[x]$ of T, determine $g_i \equiv g \bmod f_i$, where $\deg g_i < \deg f_i$ for $1 \le i \le k$. This can be done with $O(\mathrm{M}(r) + n\mathrm{M}(n))$ operations in K by first reducing g modulo f_1, since all the other invariant factors divide f_1. Next compute the matrices $g_i(C_{f_i}) = g(C_{f_i})$ for $1 \le i \le k$. Using Lemma 7.1 above, this can be accomplished with $O(n^2)$ operations in K. Now $g(S) = \mathrm{diag}(g(C_{f_1}), \ldots, g(C_{f_k}))$ and $g(T) = Ug(S)U^{-1}$. □

A more general theorem can be obtained using arithmetic circuits or straight-line programs (see Aho, Hopcroft, and Ullman (1974), §1.5 or von zur Gathen (1986) for a formal definition).

This model gives a very natural computational representation of rational functions, nicely formalizing and generalizing such notions as sparseness and ease of computation. Briefly, an algebraic circuit or straight-line program $\wp$ over a field K consists of a finite list $s_1, s_2, \ldots, s_m$ of statements, each of the form $a_i := b_i \operatorname{op}_i c_i$, where a_i is a variable (assigned only at step i), $\operatorname{op}_i \in \{+, -, \times, /\}$, and each of b_i, c_i is either equal to a_j for some $1 \leq j < i$, a constant in K, or the *input* x. The output is the value of a_m upon evaluation of each of the steps in sequence (this semantics glosses over a number of subtleties—see von zur Gathen (1986)). The sequential cost of $\wp$ is m. To define the parallel cost, note that $\wp$ defines a directed acyclic graph on the nodes $x, s_1, \ldots, s_k$, where there is an edge from s_i to s_j (or x to s_j) if s_j references a_i (resp., x). The parallel cost is the depth or maximum path length from the node x to the node s_m.

Now let $\wp$ be an algebraic circuit over K. Let F be an extension ring of K, so $\wp$ can be thought of as computing a function from $\mathsf{F} \to \mathsf{F}$. If $\wp$ has length m and depth d, then we can evaluate $\wp$ at a point $a \in \mathsf{F}$ on a sequential arithmetic RAM over F with $O(m)$ operations in F, or on an arithmetic PRAM over F in time $O(d)$ on $O(m)$ processors. For straight-line programs we obtain a theorem similar to Theorem 7.2.

THEOREM 7.3. *Let $\wp$ be a straight-line program over K as described above with length m and depth d. Assume that K has characteristic p and $\#\mathsf{K} \geq n^2$. For any $T \in \mathsf{K}^{n \times n}$, we can evaluate $\wp$ at T in time as follows:*

(i) *with $O(\mathrm{MM}(n) \log n + (n+m)\mathrm{M}(n))$ operations in K sequentially;*

(ii) *in parallel time $O(\log^4 n + d \log n)$ when $p \geq n$ or $p = 0$, or $O(\log^5 n / \log p + d \log n)$ when $2 \leq p < n$, on $O(\mathrm{MM}(n) \log n + (n+m)\mathrm{M}(n))$ processors.*

The algorithm is of the Las Vegas type, returns the correct answer with probability at least $1/4$, and otherwise reports "failure."

Proof. First compute the Frobenius form $S \in \mathsf{K}^{n \times n}$ of T, and an invertible $U \in \mathsf{F}^{n \times n}$ such that $U^{-1} T U = S$. Assume that $S = \operatorname{diag}(C_{f_1}, \ldots, C_{f_k})$, where $f_1, \ldots, f_k \in \mathsf{K}[x]$ are the invariant factors of T, and $C_{f_1}, \ldots, C_{f_k}$ are their respective companion matrices. Then evaluate $\wp$ at $x \bmod f_1$ in the ring $\mathsf{K}[x]/(f_1)$. If this evaluation fails, then we report that the circuit is undefined at T. Otherwise, assume the result of this evaluation yields $h \bmod f_1$ for some $h \in \mathsf{K}[x]$. Next compute $h_i \equiv h \bmod f_i$ for $1 \leq i \leq k$. The final result is then $U^{-1} \operatorname{diag}(h_1(C_{f_1}), \ldots, h_k(C_{f_k}))U \in \mathsf{K}^{n \times n}$. Correctness is clear from the ring-isomorphism between $\mathsf{K}[T]$ and $\mathsf{K}[x] \bmod f_1$.

The cost of the algorithm is measured as follows. Computing the Frobenius form requires $O(\mathrm{MM}(n) \log n + n\mathrm{M}(n))$ operations in K using Theorem 5.1. In parallel, we can use Theorem 6.4, which requires $O(\log^4 n)$ time on $O(\mathrm{MM}(n))$ processors when $p \geq n$ or $p = 0$, or time $O(\log^5 n / \log p)$ on $O(\mathrm{MM}(n))$ processors when $2 \leq p < n$. We can evaluate $\wp$ at x in $\mathsf{K}[x]/(f_1)$ with $O(m \cdot \mathrm{M}(n))$ operations in K sequentially, or time $O(d \log n)$ on $O(m \cdot \mathrm{M}(n))$ processors in parallel. The time required by the remaining steps is dominated by the time required by these first two. □

As an application of this straight-line program evaluation technique we consider computing high powers of matrices, an operation performed in some cryptographic systems (see, for example, Chuang and Dunham (1990)). Using linear recurrences and companion matrices to compute powers of matrices is certainly not new; it dates at least to Ranum (1911).

COROLLARY 7.4. *Given $s \geq 0$ and $T \in \mathsf{K}^{n \times n}$, we can compute $T^s \in \mathsf{K}^{n \times n}$ with a Las-Vegas-type probabilistic algorithm requiring $O(\mathrm{MM}(n) \log n + (n + \log s)\mathrm{M}(n))$ or $\tilde{O}(\mathrm{MM}(n) + n \log s)$ operations in K, where K is any field with at least n^2 elements.*

Proof. The repeated squaring method of computing high powers of elements in any group yields a straight-line program of depth and length $O(\log s)$. Apply Theorem 7.3 to obtain the result. □

We now summarize the above results for fields with fewer than n^2 elements. These follow immediately from the use of Theorems 5.2 (sequential) and 6.5 (parallel) instead of Theorems 5.1 and 6.4, respectively.

THEOREM 7.5. *Let* K *be any field with* $q < n^2$ *elements and characteristic* p, $T \in \mathsf{K}^{n\times n}$, *and* F *be the smallest extension field of* K *containing* n^2 *elements. Las-Vegas-type probabilistic algorithms exist as follows:*

(i) *Given* $g \in \mathsf{K}[x]$ *of degree* r, *we can evaluate* $g(T) \in \mathsf{K}^{n\times n}$ *with* $O((\mathrm{MM}(n)\log n + n\mathrm{M}(n)) \cdot \mathrm{M}(\log_q n)\log\log_q n + \mathrm{M}(r))$ *or* $O^{\sim}(\mathrm{MM}(n) + r)$ *operations in* K *sequentially. In parallel we can do this in time* $O(\log^4 n \log\log_q n + \log r)$ *when* $p = 0$ *or* $p \geq n$, *or in time* $O(\log^5 n \log\log_q n/\log p + \log r)$ *when* $2 \leq p < n$, *on* $O(\mathrm{MM}(n)\mathrm{M}(\log_q n) + \mathrm{M}(r))$ *processors.*

(ii) *Given a straight-line program* $\wp$ *with length* m *and depth* d, *we can evaluate* $\wp$ *at* T *with* $O((\mathrm{MM}(n)\log n + n\mathrm{M}(n)) \cdot \mathrm{M}(\log_q n)\log\log_q n + m\mathrm{M}(n))$ *operations in* K. *In parallel we can do this in time* $O(\log^4 n \log\log_q n + d\log n)$ *when* $p = 0$ *or* $p \geq n$, *or in time* $O(\log^5 n \log\log_q n/\log p + d\log n)$ *when* $2 \leq p < n$, *on* $O(\mathrm{MM}(n)\mathrm{M}(\log_q n) + m \cdot \mathrm{M}(n))$ *processors.*

A lower bound on evaluating polynomials at matrices. In this subsection we show that evaluating a nonlinear polynomial at a matrix is at least as hard as matrix multiplication. Specifically, if $g \in \mathsf{K}[x]$ has degree $r \geq 2$ and we can evaluate $g(T)$ at any $T \in \mathsf{K}^{n\times n}$ with at most $t(n)$ operations in K, then we can multiply two $n \times n$ matrices over K with $O(t(n))$ operations in K. We make a technical assumption about the cost function $t\colon \mathbb{R}_{>0} \to \mathbb{R}_{>0}$:

$$\forall a \in \mathbb{R}_{>0}\ \forall b \in \mathbb{R}_{\geq 1}\quad b^2 t(a) \leq t(ab) \leq b^{3.5} t(a).$$

The problem of evaluating a polynomial at a matrix has a trivial lower bound of $\Omega(n^2)$ operations in K and, as discussed above, a deterministic upper bound of $O(n^{3.5})$ operations in K, making this assumption reasonable.

We will assume here that $\#\mathsf{K} \geq r - 1$. A similar argument is presented in Giesbrecht (1993) for smaller fields. Let $A, B \in \mathsf{K}^{n\times n}$. We show how to use an algorithm for evaluating g at a matrix to multiply A and B. Consider the matrix

$$T = \begin{pmatrix} I_n & A & 0_n \\ 0_n & cI_n & B \\ 0_n & 0_n & I_n \end{pmatrix} \in \mathsf{K}^{3n\times 3n},$$

where $c \in \mathsf{K}$ and 0_n, $I_n \in \mathsf{K}^{n\times n}$ are the zero and identity matrices, respectively.

LEMMA 7.6. *For* $j \geq 1$,

$$T^j = \begin{pmatrix} I_n & \left(\sum_{0\leq i\leq j-1} c^i\right)\cdot A & \left(\sum_{0\leq i\leq j-2} (j-i-1)c^i\right)\cdot AB \\ 0_n & c^j I_n & \left(\sum_{0\leq i\leq j-1} c^i\right)\cdot B \\ 0_n & 0_n & I_n \end{pmatrix} \in \mathsf{K}^{3n\times 3n}.$$

Proof. We proceed by induction on j. Clearly the lemma is true for $j = 1$. Assume the lemma holds for $j - 1$ (we assume $j \geq 2$). Then

$$T^j = T \cdot T^{j-1} = \begin{pmatrix} I_n & A & 0_n \\ 0_n & cI_n & B \\ 0_n & 0_n & I_n \end{pmatrix} \begin{pmatrix} I_n & \left(\sum_{0 \leq i \leq j-2} c^i\right) \cdot A & \left(\sum_{0 \leq i \leq j-3} (j-i-2)\, c^i\right) \cdot AB \\ 0_n & c^{j-1} I_n & \left(\sum_{0 \leq i \leq j-2} c^i\right) \cdot B \\ 0_n & 0_n & I_n \end{pmatrix}$$

$$= \begin{pmatrix} I_n & \left(\sum_{0 \leq i \leq j-1} c^i\right) \cdot A & \left(\sum_{0 \leq i \leq j-2} (j-i-1)\, c^i\right) \cdot AB \\ 0_n & c^j I_n & \left(\sum_{0 \leq i \leq j-1} c^i\right) \cdot B \\ 0_n & 0_n & I_n \end{pmatrix}. \qquad \square$$

Let $C \in \mathsf{K}^{n \times n}$ be the top right $n \times n$ block of $g(T)$. If $g = \sum_{0 \leq i \leq r} b_i x^i$, then

$$C = \sum_{0 \leq i \leq r} b_i \sum_{0 \leq j \leq i-2} c^j AB = \left(\sum_{0 \leq i \leq r-2} c^j \left(\sum_{i+2 \leq j \leq r} (j-i-1) b_i\right)\right) \cdot AB = h(c) \cdot AB,$$

where $h = \sum_{0 \leq i \leq r-2} z^i \left(\sum_{i+2 \leq j \leq r} (j-i-1) b_j\right) \in \mathsf{K}[z]$ and z is an indeterminate. Since the coefficient of z^{r-2} in h is $b_r \neq 0$, we know $\deg h = r - 2$. Now, we have assumed that $\#\mathsf{K} > r - 2$, so there exists a $c \in \mathsf{K}$ such that $h(c) \neq 0$ because h has at most $r - 2$ roots. Such a c can be found by evaluating h at $r - 1$ distinct points in K, and using fast multipoint evaluation (see Aho, Hopcroft, and Ullman (1974), §8.5), it can be found with $O(\mathrm{M}(r) \log r)$ operations in K. Now $AB = h(c)^{-1} C$, which can be computed with an additional $O(t(n))$ operations in K.

THEOREM 7.7. *Let $g \in \mathsf{K}[x]$ have degree $r \geq 2$. Suppose we can evaluate $g(T)$ for any matrix $T \in \mathsf{K}^{n \times n}$ with $t(n)$ operations in K. If $\#\mathsf{K} \geq r - 1$ we can multiply two $n \times n$ matrices with $O(t(n) + \mathrm{M}(r) \log r)$ operations in K.*

In fact, for any given n, we need only compute a single $c \in \mathsf{K}$ such that $h(c) \neq 0$. After such a c has been found, evaluating g at any matrix in $\mathsf{K}^{n \times n}$ requires only $O(t(n))$ operations in K.

8. Computing the rational Jordan form. The rational Jordan form of a matrix $T \in \mathsf{K}^{n \times n}$ is a generalization of the usual Jordan form. Whereas a Jordan form of T exists only if K contains all eigenvalues of T, the rational Jordan form is a matrix in $\mathsf{K}^{n \times n}$. The eigenvalues of T, which may generate an algebraic extension of K of exponential degree over K, are replaced by the companion matrices of their minimal polynomials in $\mathsf{K}[x]$. For any monic irreducible $g \in \mathsf{K}[x]$ of degree r and any $m > 0$, define the *rational Jordan block* $J_g^{(m)} \in \mathsf{K}^{mr \times mr}$ as

$$J_g^{(m)} = \begin{pmatrix} \boxed{C_g}\ \boxed{I_r} & & & \\ & \boxed{C_g}\ \boxed{I_r} & & \\ & & \ddots & \ddots \\ & & & \boxed{I_r} \\ & & & \boxed{C_g} \end{pmatrix} \in \mathsf{K}^{mr \times mr}, \tag{8.1}$$

where I_r is the $r \times r$ identity matrix and $C_g \in \mathsf{K}^{r \times r}$ is the companion matrix of g. It is easily shown that both the minimal and characteristic polynomials of $J_g^{(m)}$ are equal to $g^m \in \mathsf{K}[x]$ (see Lemma 8.1 below). Any $T \in \mathsf{K}^{n \times n}$ is similar to its *rational Jordan form*, a matrix $Q = \operatorname{diag}(Q_1, Q_2, \ldots, Q_l) \in \mathsf{K}^{n \times n}$, where $Q_1, \ldots, Q_l$ are rational Jordan blocks. Note that when all eigenvalues of T lie in K, then Q is in fact the usual Jordan form of T. The rational Jordan form is unique up to the order of the (rational) Jordan blocks, as is the usual Jordan form. Rational forms akin to the rational Jordan form were investigated at least as early as Frobenius (1911). Kaltofen, Krishnamoorthy, and Saunders (1990) exhibit fast probabilistic parallel algorithms for a somewhat different rational Jordan form, as well as a "symbolic Jordan form" of a matrix. Roch and Villard (1994) demonstrate fast deterministic parallel algorithms, in the complexity class NC^3, for the symbolic Jordan form of a matrix.

Let K be any field with at least n^2 elements. The problem of finding the rational Jordan form $Q \in \mathsf{K}^{n \times n}$ of $T \in \mathsf{K}^{n \times n}$ is reducible, with $O(\mathrm{MM}(n) \log n + n\mathrm{M}(n))$ operations in K, to factoring a single polynomial in $\mathsf{K}[x]$ of degree at most n, namely, the minimal polynomial of T which is found from the Frobenius form of T. The necessary observation is that once we know the Frobenius form of T and the complete factorization of its minimal polynomial f_1, we can immediately determine the rational Jordan form of T.

LEMMA 8.1. *Let $g \in \mathsf{K}[x]$ be monic and irreducible of degree r, and $m \geq 1$. The rational Jordan block $J_g^{(m)} \in \mathsf{K}^{mr \times mr}$ has minimal polynomial g^m.*

Proof. If $C_g \in \mathsf{K}^{r \times r}$ is the companion matrix of g, then $\mathsf{K}[C_g]$ is a field isomorphic to $\mathsf{L} = \mathsf{K}[z]/(g)$ for an indeterminate z (see Lidl and Niederreiter (1983), §2.5). This isomorphism is realized by mapping $C_g \mapsto z \bmod g$ and can be extended to an isomorphism of matrices in $\mathsf{K}[C_g]^{m \times m}$ with $\mathsf{L}^{m \times m}$. Observe that $J_g^{(m)} \in \mathsf{K}[C_g]^{m \times m}$ and its image $\mathcal{J}_g^{(m)} \in \mathsf{L}^{m \times m}$ is a Jordan block with its eigenvalue $\gamma = z \bmod g$ on the diagonal and ones on the superdiagonal. The minimal polynomial of $\mathcal{J}_g^{(m)}$ is $(x - \gamma)^m \in \mathsf{L}[x]$, so the minimal polynomial of $J_g^{(m)}$ in $\mathsf{K}[x]$ is the multiple of $(x - \gamma)^m$ in $\mathsf{K}[x] \setminus \{0\}$ of lowest degree. This is precisely g^m. □

LEMMA 8.2. *Let $f \in \mathsf{K}[x]$ be monic of degree n and $C_f \in \mathsf{K}^{n \times n}$ be the companion matrix of f. If $f = g_1^{m_1} \ldots g_s^{m_s}$ for distinct monic irreducible $g_1, \ldots, g_s \in \mathsf{K}[x]$ and $m_1, \ldots, m_s > 0$, then C_f has rational Jordan form $B = \operatorname{diag}(J_{g_1}^{(m_1)}, \ldots, J_{g_s}^{(m_s)})$.*

Proof. It is well known that the minimal polynomial of a block diagonal matrix is the least common multiple of the minimal polynomials of its blocks. Thus, $\min(B; \mathsf{K}^{n \times 1}) = \operatorname{lcm}(g_1^{e_1}, \ldots, g_s^{e_s}) = g_1^{e_1} \ldots g_s^{e_s} = f$, since the minimal polynomial of $J_{g_i}^{(e_i)}$ is $g_i^{m_i}$ for $1 \leq i \leq s$ by Lemma 8.1. Since $\deg f = n$ and f is the minimal polynomial of B, B is similar to the companion matrix C_f of f. □

THEOREM 8.3. *Suppose $T \in \mathsf{K}^{n\times n}$ has invariant factors $f_1, \ldots, f_k \in \mathsf{K}[x]$, where f_i has companion matrix $C_{f_i} \in \mathsf{K}^{d_i\times d_i}$, with* $\deg f_i = d_i$ *for* $1 \le i \le k$. *If $B_i \in \mathsf{K}^{d_i\times d_i}$ is the rational Jordan form of C_{f_i} for $1 \le i \le k$, then $Q = \operatorname{diag}(B_1, \ldots, B_k) \in \mathsf{K}^{n\times n}$ is the rational Jordan form of T.*

Proof. By Lemma 8.2 there exists a nonsingular $P_i \in \mathsf{K}^{d_i\times d_i}$ such that $B_i = P_i^{-1} C_{f_i} P_i$ for $1 \le i \le k$. The matrix $P = \operatorname{diag}(P_1, \ldots, P_k) \in \mathsf{K}^{n\times n}$ satisfies $Q = P^{-1}TP$, and Q is in rational Jordan form. □

THEOREM 8.4. *Let K be any field with at least n^2 elements. Given $T \in \mathsf{K}^{n\times n}$, there is a Las Vegas reduction from the problem of finding the rational Jordan form $Q \in \mathsf{K}^{n\times n}$ of T and an invertible $P \in \mathsf{K}^{n\times n}$ such that $Q = P^{-1}TP$ to the problem of factoring a single polynomial in $\mathsf{K}[x]$ of degree at most n (namely, the minimal polynomial $f_1 \in \mathsf{K}[x]$ of T). This reduction requires $O(\mathrm{MM}(n)\log n + n\mathrm{M}(n))$ or $O^{\sim}(\mathrm{MM}(n))$ operations in K.*

Proof. We find the Frobenius form $S \in \mathsf{K}^{n\times n}$ of T and an invertible $U_0 \in \mathsf{K}^{n\times n}$ such that $U_0^{-1} T U_0 = S$ using the algorithm `FrobeniusForm`. We factor the first invariant factor $f_1 \in \mathsf{K}[x]$ (determined by the Frobenius form and identical to the minimal polynomial of T) and construct the rational Jordan form $Q \in \mathsf{K}^{n\times n}$ of T, using Theorem 8.3. An invertible $U_1 \in \mathsf{K}^{n\times n}$ such that $U_1^{-1} Q U_1 = S$ is constructed using `FrobeniusForm` (we know Q is similar to S by its construction). Then $Q = U_1 U_0^{-1} T U_0 U_1^{-1}$ and $P = U_0 U_1^{-1}$ satisfies $Q = P^{-1}TP$. □

To find the rational Jordan form of $T \in \mathsf{K}^{n\times n}$ when K has fewer than n^2 elements, we embed K into a slightly larger field F to find the Frobenius form of T, using Theorem 5.2. As in Theorem 5.2, $\mathsf{F} = \mathsf{K}[x]/(\psi)$, where $\psi \in \mathsf{K}[x]$ is monic and irreducible of degree $\lceil 2\log_q n \rceil$.

THEOREM 8.5. *Let K be a field with $q = \#\mathsf{K} < n^2$. Given $T \in \mathsf{K}^{n\times n}$, there is a Las Vegas reduction from the problem of finding the rational Jordan form $Q \in \mathsf{K}^{n\times n}$ of T and an invertible $P \in \mathsf{F}^{n\times n}$, such that $Q = P^{-1}TP$, to factoring a single polynomial in $\mathsf{K}[x]$ of degree at most n (namely, the minimal polynomial $f_1 \in \mathsf{K}[x]$ of T). This reduction requires $O((\mathrm{MM}(n)\log n + n\mathrm{M}(n)) \cdot \mathrm{M}(\log_q n)\log\log_q n)$ or $O^{\sim}(\mathrm{MM}(n))$ operations in K.*

Using the Las Vegas algorithm of Berlekamp (1970) for factoring polynomials over finite fields, we obtain an algorithm for computing the rational Jordan form of a matrix, given only that matrix.

COROLLARY 8.6. *Let K be a finite field of size q and $T \in \mathsf{K}^{n\times n}$. The rational Jordan form of T can be computed by a Las Vegas algorithm with $O(\mathrm{MM}(n)\log n + n\mathrm{M}(n) + n\log q)$ or $O^{\sim}(\mathrm{MM}(n) + n\log q)$ operations in K if $\#\mathsf{K} \ge n^2$, and $O((\mathrm{MM}(n)\cdot\log n + n\mathrm{M}(n)) \cdot \mathrm{M}(\log_q n)\log\log_q n + n\log q)$ or $O^{\sim}(\mathrm{MM}(n) + n\log q)$ operations in K if $\#\mathsf{K} < n^2$.*

Acknowledgment. The author thanks Joachim von zur Gathen and Erich Kaltofen for their enthusiasm and help while writing this paper.

REFERENCES

A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN (1974), *The Design and Analysis of Computer Algorithms*, Addison–Wesley, Reading, MA.

D. AUGOT AND P. CAMION (1993), *The minimal polynomials, characteristic supspaces, normal bases and the Erobenius form*, Tech. report 2006, Unité de Recherche INRIA Rocquencourt, France.

W. BAUR AND V. STRASSEN (1982), *The complexity of partial derivatives*, Theoret. Comput. Sci., 22, pp. 317–330.

E. R. BERLEKAMP (1970), *Factoring polynomials over large finite fields*, Math. Comp., 24, pp. 713–735.

D. BINI AND V. PAN (1992), *Improved parallel polynomial division*, in Proc. 33rd IEEE Symposium on Foundations of Computer Science, Pittsburgh, PA, pp. 131–136.

A. BORODIN AND I. MUNRO (1975), *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, New York, NY.

R. P. BRENT AND H. T. KUNG (1978), *Fast algorithms for manipulating formal power series*, J. Assoc. Comput. Mach., 25, pp. 581–595.

J. BUNCH AND J. HOPCROFT (1974), *Triangular factorization and inversion by fast matrix multiplicaton*, Math. Comp., 28, pp. 231–236.

D. CANTOR AND E. KALTOFEN (1991), *Fast multiplication of polynomials over arbitrary algebras*, Acta Inform., 28, pp. 693–701.

C. CHUANG AND J. DUNHAM (1990), *Matrix extensions of the RSA algorithm*, in Proc. Crypto 1990, pp. 137–153.

D. COPPERSMITH AND S. WINOGRAD (1990), *Matrix multiplication via arithmetic progressions*, J. Symbolic Comput., 9, pp. 251–280.

A. DANILEVSKY (1937), *The numerical solution of the secular equation*, Mat. Sb., 44, pp. 169–171. (In Russian.)

W. EBERLY (1991), *Efficient parallel independent subsets and matrix factorizations*, in Proc. 3rd IEEE Symposium on Parallel and Distributed Processing, Dallas, TX, pp. 204–211.

D. FADDEEV AND V. FADDEEVA (1963), *Computational methods of linear algebra*, W. H. Freeman, San Francisco, CA.

G. FROBENIUS (1911), *Über den Rang einer Matrix*, Sitzungsberichte der Königlich Preußischen Akademie der Wissenschaften zu Berlin, pp. 54–65.

F. R. GANTMACHER (1990), *The Theory of Matrices, Vol.* I, Chelsea, New York, NY.

J. VON ZUR GATHEN (1986), *Parallel arithmetic computations: A survey*, in Proc. 12th International Symposium on Math. Foundations of Computer Science, Lecture Notes in Computer Science 233, Springer-Verlag, Bratislava, pp. 93–112.

——— (1993), *Parallel linear algebra*, in Synthesis of Parallel Algorithms, J. Reif, ed., Morgan Kaufmann, San Mateo, CA, pp. 573–617.

M. GIESBRECHT (1993), *Nearly Optimal Algorithms for Canonical Matrix Forms*, Ph.D. thesis, Department of Computer Science, University of Toronto.

K. HOFFMAN AND R. KUNZE (1971), *Linear Algebra*, Prentice–Hall, Englewood Cliffs, NJ.

E. KALTOFEN (1992), *Efficient solution of sparse linear systems*, Lecture notes, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY.

E. KALTOFEN, M. S. KRISHNAMOORTHY, AND B. D. SAUNDERS (1987), *Fast parallel computation of Hermite and Smith forms of polynomial matrices*, SIAM J. Algebraic Discrete Meth., 8, pp. 683–690.

——— (1990), *Parallel algorithms for matrix normal forms*, Linear Algebra Appl., 136, pp. 189–208.

E. KALTOFEN AND V. PAN (1991), *Processor-efficient parallel solution of linear systems over an abstract field*, in Proc. 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 180–191.

——— (1992), *Processor-efficient parallel solution of linear systems II: The general case*, in Proc. 33rd IEEE Symposium on Foundations of Computer Science, Pittsburgh, PA, pp. 714–723.

R. KANNAN (1985), *Polynomial-time algorithms for solving systems of linear equations over polynomials*, Theoret. Comput. Sci., 39, pp. 69–88.

R. KANNAN AND A. BACHEM (1979), *Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix*, SIAM J. Comput., 8, pp. 499–507.

R. M. KARP AND V. RAMACHANDRAN (1990), *Parallel algorithms for shared-memory machines*, in Handbook of Theoretical Computer Science, Vol. A, Algorithms and Complexity, J. van Leeuwen, ed., MIT Press/Elsevier, Cambridge, MA, pp. 869–932.

W. KELLER-GEHRIG (1985), *Fast algorithms for the characteristic polynomial*, Theoret. Comput. Sci., 36, pp. 309–317.

R. LIDL AND H. NIEDERREITER (1983), *Finite Fields*, Encyclopedia of Mathematics and its Applications, Vol. 20, Addison–Wesley, Reading MA.

H. LÜNEBURG (1987), *On Rational Normal Form of Endomorphisms: A Primer to Constructive Algebra*, Wissenschaftsverlag, Mannheim.

H. J. NUSSBAUMER (1980), *Fast polynomial transform algorithms for digital convolutions*, IEEE Trans. Acoustics, Speech and Signal Processing, 28, pp. 395–398.

P. OZELLO (1987), *Calcul Exact Des Formes De Jordan et de Frobenius d'une Matrice*, Ph.D. thesis, Université Scientifique Technologique et Medicale de Grenoble.

V. PAN AND J. REIF (1985), *Efficient parallel solutions of linear systems*, in Proc. 17th Annual Symposium on Theory of Computing, Providence, RI, pp. 143–152.

M. S. PATERSON AND L. STOCKMEYER (1973), *On the number of nonscalar multiplications necessary to evaluate polynomials*, SIAM J. Comput., 2, pp. 60–66.

A. RANUM (1911), *The general term of a recurring series*, Bull. Amer. Math. Soc., 17, pp. 457–461.

J. ROCH AND G. VILLARD (1994), *Calcul formel et parallélisme: étude de la forme normal de Jordan*, Tech. Report 7, Institut IMAG, Laboratoire LMC, Université Joseph Fourier, Genoble, France.

A. SCHÖNHAGE (1977), *Schnelle Multiplikation von Polynomen über Körpern der Charakteristik* 2, Acta Inform., 7, pp. 395–398.

A. SCHÖNHAGE AND V. STRASSEN (1971), *Schnelle Multiplikation großer Zahlen*, Computing, 7, pp. 281–292.

J. T. SCHWARTZ (1980), *Fast probabilistic algorithms for verification of polynomial identities*, J. Assoc. Comput. Mach., 27, pp. 701–717.

V. SHOUP (1993), *Fast construction of irreducible polynomials over finite fields*, in Proc. 4th Annual ACM–SIAM Symposium on Discrete Algorithms, Austin, TX, pp. 484–492.

B. L. VAN DER WAERDEN (1970), *Algebra*, 7th ed., Vol. 1, Frederick Ungar, New York.

G. VILLARD (1994), *Calcul formel et parallélisme: étude de la forme normal de Smith*, Tech. report 8, Institut IMAG, Laboratoire LMC, Université Joseph Fourier, Genoble, France.

D. WIEDEMANN (1986), *Solving sparse linear equations over finite fields*, IEEE Trans. Inform. Theory, IT-32, pp. 54–62.

Y. ZALCSTEIN AND M. GARZON (1987), *An NC^2 algorithm for testing similarity of matrices*, Inform. Process. Lett., pp. 253–254.

SIAM J. COMPUT.
Vol. 24, No. 5, pp. 970–1001, October 1995

003

DYNAMIC GRAPH DRAWINGS: TREES, SERIES-PARALLEL DIGRAPHS, AND PLANAR *ST*-DIGRAPHS*

ROBERT F. COHEN†, GIUSEPPE DI BATTISTA‡, ROBERTO TAMASSIA§, AND IOANNIS G. TOLLIS¶

Abstract. Drawing graphs is an important problem that combines elements of computational geometry and graph theory. Applications can be found in a variety of areas including circuit layout, network management, software engineering, and graphics.

The main contributions of this paper can be summarized as follows:

- We devise a model for dynamic graph algorithms, based on performing queries and updates on an implicit representation of the drawing, and we show its applications.
- We present efficient dynamic drawing algorithms for trees and series-parallel digraphs.

As further applications of the model, we give dynamic drawing algorithms for planar *st*-digraphs and planar graphs. Our algorithms adopt a variety of representations (e.g., straight line, polyline, visibility) and update the drawing in a smooth way.

Key words. graph drawing, dynamic algorithms, data structures, layout, trees, series-parallel digraphs, planar graphs

AMS subject classifications. 68Q20, 68P05, 05C85, 65Y25

1. Introduction.

Drawing graphs is an important problem that combines elements of computational geometry and graph theory. Applications can be found in a variety of areas including circuit layout, network management, software engineering, and graphics. For a survey on graph drawing, see [7]. While this area has recently received increasing attention (see, e.g., [3], [11], [17], [18], [24], [27], [35]), the study of drawing graphs in a dynamic setting has been an open problem. Previous work [28] only considers trees and presents a technique that restructures the drawing of a tree in time proportional to its height, which is linear in the worst case.

The motivation for investigating dynamic graph drawing algorithms arises when very large graphs need to be visualized in a dynamic environment, where vertices and edges are inserted and deleted and subgraphs are displayed. Several graph manipulation systems allow the user to interactively modify a graph; hence, techniques that support fast restructuring of the drawing would be very useful. Also, it is important that the dynamic drawing algorithm not alter drastically the structure of the drawing after a local modification of the graph. In fact, human interaction requires a "smooth" evolution of the drawing.

*Received by the editors July 15, 1992; accepted for publication (in revised form) March 28, 1994. This research was supported in part by the National Science Foundation under grants CCR-9007851 and CCR-9423847; by the U.S. Army Research Office under grants DAAL03-91-G-0035 and DAAH04-93-0134; by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-91-J-4052, ARPA order 8225; by Cadre Technologies, Inc.; by the Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo of the Italian National Research Council; and by the Esprit II BRA of the European Community (project ALCOM). An extended abstract of this paper was presented at the Eighth ACM Symposium on Computational Geometry, Berlin, 1992. This work was performed in part while the authors were visiting, the Istituto di Analisi dei Sistemi ed Informatica of the Italian National Research Council (IASI–CNR).

†Department of Computer Science, University of Newcastle, University Drive, Callaghan, New South Wales 2308, Australia (rfc@cs.newcastle.edu.au). The work of this author was performed in part while he was at Brown University, Providence, Rhode Island.

‡Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza," Via Salaria 113, 00198 Rome, Italy (dibattista@iasi.rm.cnr.it). The work of this author was performed in part while he was visiting the International Computer Science Institute, Berkeley, California.

§Department of Computer Science, Brown University, Providence, Rhode Island 02912-1910 (rt@cs.brown.edu).

¶Department of Computer Science, The University of Texas at Dallas, Richardson, Texas 75083-0688 (tollis@utdallas.edu).

In this paper we present dynamic algorithms for drawing planar graphs under a variety of drawing standards. We consider straight-line, polyline, grid, upward, and visibility drawings together with aesthetic criteria that are important for readability, such as the display of planarity, symmetry, and reachability. Also, we provide techniques that are especially tailored for important subclasses of planar graphs such as trees and series-parallel digraphs. Our dynamic drawing algorithms have the important property of performing "smooth updates" of the drawing. In what follows we use n and m to denote the number of vertices and edges of a given graph.

1.1. Definitions. A *drawing* Γ of a graph G maps each vertex of G to a distinct point of the plane and each edge (u, v) of G to a simple Jordan curve with endpoints u and v. We say that Γ is a *straight-line* drawing if each edge is a straight-line segment; Γ is a *polyline* drawing if each edge is a polygonal chain; Γ is an *orthogonal* drawing if each edge is a chain of alternating horizontal and vertical segments. A *grid* drawing is such that the vertices and bends along the edges have integer coordinates. *Planar* drawings, where edges do not intersect, are especially important because they improve the readability of the drawing. A *planar embedding* specifies the circular order of the edges around a vertex in a planar drawing. Hence, different drawings may have the same planar embedding. Note that a planar graph may have an exponential number of planar embeddings (see, e.g., [29]). An *upward* drawing of an acyclic digraph has all the edges flowing from bottom to top. Planar upward drawings are attracting increasing theoretical and practical interest [3], [8], [9], [11], [15], [24], [38], [48]. A *visibility representation* maps vertices to horizontal segments and edges to vertical segments that intersect only the two corresponding vertex segments.

We assume the existence of a *resolution rule* that implies a finite minimum area for the drawing of a graph. Two typical resolution rules are integer coordinates for the vertices, or a minimum distance δ between any two vertices. When a resolution rule is given, it is useful to consider the problem of finding drawings with minimum area. Planar drawings require $\Omega(n^2)$ area in the worst case [12]. Further results on the area of planar drawings appear in [2], [11], [18], [35].

1.2. Model. Here we describe a framework for dynamic graph drawing algorithms. At a first glance, it appears that updating a drawing may require $\Omega(n+m)$ time in the worst case, since we may have to change the coordinates of all vertices and edges. Our approach is to consider graph drawing problems in a "query/update" setting. Namely, we aim for maintaining an *implicit* representation of the drawing of a graph G such that the following operations can be efficiently performed:

- *Drawing queries* that return the drawing of a subgraph S of G consistent with the overall drawing of G. We aim for an output-sensitive time complexity for this operation, i.e., a polynomial in $\log n$ and k, where k is the size of S. Ideally, the time complexity should be $O(k+\log n)$. A special case of this query ($S = \{v\}$) returns the coordinates of a single vertex v.
- *Window queries* that return the portion of the drawing inside a query rectangle.
- *Point-location queries* in the subdivision of the plane induced by the drawing of G. Such queries are defined when the drawing of G is planar.
- *Update operations*, e.g., insertion and deletion of vertices and edges or replacement of an edge by a graph, which modify the (implicit) representation of the drawing accordingly.

There are two types of quality measures in dynamic graph drawing: the "aesthetic" properties of the drawing being maintained and the space–time complexity of queries and updates. There is an inherent trade-off between the two. For example, it is very easy to maintain the drawing of a graph where the vertices are randomly placed on the plane and the edges are drawn as straight-line segments. However, the aesthetic quality of the drawings produced by this simple strategy is typically not satisfactory. On the other hand, if we want to

guarantee optimal drawings with respect to some aesthetic criteria, e.g., planarity, symmetry, etc., the update/query operations may require high time complexity. In what follows we present techniques with polylogarithmic query/update time that maintain drawings that are optimal with respect to a set of aesthetic criteria.

More formally, a dynamic graph drawing problem consists of

- A class of graphs $\mathcal{G}$ to be drawn.
- A repertory $\mathcal{O}$ of operations to be performed, subdivided into
  - A set $\mathcal{Q}$ of *query operations* (such as drawing, window, and point location queries) that ask questions on the drawing of the current graph.
  - A set $\mathcal{U}$ of *update operations* that modify the current graph and restructure its drawing, such as insertion and deletion of vertices and edges.

The drawing is modified only by update operations and is not changed by queries.

- A *static drawing predicate* $\mathcal{P}_S$ that expresses "aesthetic" properties to be satisfied by the drawing of the current graph. An example of a static drawing predicate for planar graphs is "the drawing is planar, polyline, grid, with $O(n^2)$ area, and at most $2n + 4$ bends along the edges."
- A *dynamic drawing predicate* $\mathcal{P}_D$ that expresses "similarity" properties to be satisfied by the drawings before and after an update operation. An example of a dynamic drawing predicate for trees is "the drawing of a subtree not affected by the update stays the same up to a translation." Such predicates can be used to guarantee a "smooth" evolution of the drawing.

A solution to a dynamic graph drawing problem is an algorithm that dynamically maintains a drawing of a graph of class $\mathcal{G}$ satisfying predicates $\mathcal{P}_S$ and $\mathcal{P}_D$, under a sequence of operations of repertory $\mathcal{O}$. Performance measures for the algorithm are the memory space used by the data structures and the time complexity of the various operations. Typically, there is a trade-off between the efficiency of the algorithm and the tightness of the requirements expressed by the drawing predicates $\mathcal{P}_S$ and $\mathcal{P}_D$.

1.3. Overview. The rest of this paper is organized as follows. Our main results are presented in §§2 and 3. In §2 we describe a dynamic technique for upward drawings of rooted trees. The data structure uses $O(n)$ memory space and supports updates and point-location queries in $O(\log n)$ time. Drawing queries take time $O(k + \log n)$ for a subtree and $O(k \log n)$ for an arbitrary subgraph. Window queries take time $O(k \log n)$. The drawings follow the usual convention of horizontally aligning vertices of the same level. Symmetries and isomorphisms of subtrees are displayed and the area is $O(n^2)$.

In §3 we present an algorithm for dynamically drawing series-parallel digraphs. It uses $O(n)$ memory space and supports updates in $O(\log n)$ time. Drawing queries take time $O(k + \log n)$ for a series-parallel subgraph and $O(k \log n)$ for an arbitrary subgraph. Point location queries take $O(\log n)$ time. Window queries take $O(k \log^2 n)$ time. The algorithm constructs upward straight-line drawings with $O(n^2)$ area, which is optimal in the worst case.

Further developments are presented in §4, where we give a family of algorithms that maintain various types of drawings for planar *st*-digraphs, including polyline upward drawings, and visibility representations. The drawings occupy $O(n^2)$ area, which is optimal in the worst case. All of the algorithms use $O(n)$ memory space and support updates in $O(\log n)$ time. Also, we consider (undirected) biconnected planar graphs. We present semidynamic algorithms for maintaining polyline drawings and visibility representations. The data structure uses $O(n)$ memory space and supports insertion in $O(\log n)$ amortized time (worst case for insertions that preserve the embedding). Drawing queries take $O(k \log n)$ time.

2. Dynamic tree drawing. In this section, we investigate the dynamic drawing of a rooted ordered tree T. We assume that edges are directed from the child to the parent, and we use T_μ to denote the subtree rooted at μ. To simplify formulas, in this section we assume

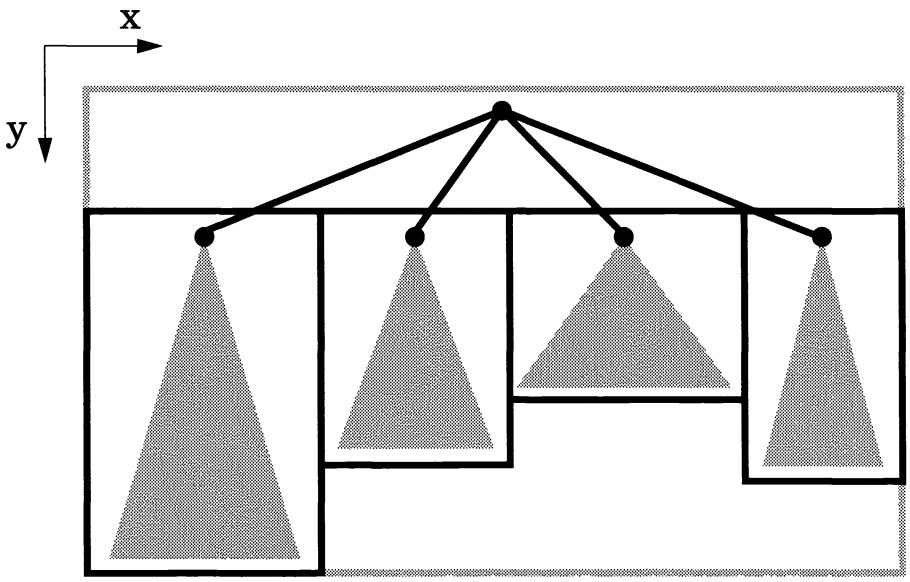


FIG. 1. *Geometric constructions in the* □*-algorithm.*

that the x-axis is directed from left to right and the y-axis is directed from top to bottom (see Fig. 1). The coordinates of a node μ in the drawing will be denoted by $x(\mu)$ and $y(\mu)$.

2.1. □-drawings. We consider the following static drawing predicate $\mathcal{P}_S$:

Upward: The drawing is upward.

Planar: The drawing is planar.

Grid: Vertices are placed at integer coordinates.

Straight-line: Edges are drawn as straight line segments.

Layered: Nodes of the same depth (distance from the root) are drawn on the same horizontal line.

Order-preserving: The left-to-right ordering of the children of a node is preserved in the drawing.

Centered: A node μ is "centered" over its children $\mu_1, \ldots, \mu_k$. Examples of variations of this rule are

- $x(\mu) = \frac{1}{k} \cdot \sum_{i=1}^{k} x(\mu_i)$;
- $x(\mu) = \frac{1}{2} \cdot (x(\mu_1) + x(\mu_k))$.

Isomorphic: Isomorphic subtrees have drawings that are congruent up to a translation.

Symmetric: Symmetric subtrees have drawings that are congruent up to a translation and a reflection.

Quadratic-area: The drawing has $O(n^2)$ area.

Reingold and Tilford [32] argue that drawings satisfying $\mathcal{P}_S$ are aesthetically pleasing and show how to construct them in $O(n)$ time. We give a fully dynamic algorithm for constructing such drawings. However, in general the drawings produced by the algorithm of [32] are less wide than those produced by our algorithm. Note that finding drawings of minimum width that satisfy the above properties is NP-hard [39].

The □-drawing of T and the *bounding box* $\square(\mu)$ of a node μ of T are recursively defined as follows (see Fig. 1):

- μ is leaf: $\square(\mu)$ is a 2×1 rectangle.

• μ has children $\mu_1, \ldots, \mu_k$: The width of $\square(\mu)$ is the sum of the widths of $\square(\mu_i)$, $1 \leq i \leq k$. The height of $\square(\mu)$ is one plus the maximum of the heights of $\square(\mu_i)$, $1 \leq i \leq k$. The bounding boxes of the children of μ are placed inside $\square(\mu)$ such that they do not overlap, their top sides are placed one unit below the top side of $\square(\mu)$, and their left-to-right order preserves the ordering of the tree.

• If μ is a leaf, it is drawn in the middle of the top side of $\square(\mu)$. Otherwise μ is drawn along the top side of $\square(\mu)$ according to the centering rule of predicate $\mathcal{P}_S$. We call the top left corner of $\square(\mu)$ the *reference point*.

To fully specify the $\square$-drawing, we assume that the reference point of the bounding box of the root of T is placed at (0, 0).

LEMMA 2.1. *Given an n-node tree T, the $\square$-drawing of T satisfies the static drawing predicate $\mathcal{P}_S$ and can be constructed in $O(n)$ time.*

Proof. It is immediate to verify that $\square$-drawings satisfy $\mathcal{P}_S$. Concerning the area, let g be the number of leaves of T and let h be the height of T. The area of the drawing is $g \cdot (h+1)$, which is $O(n^2)$. To construct the $\square$-drawing of T we use two traversals. The first traversal computes in post-order the sizes of the bounding boxes of the subtrees of T. The second traversal computes in pre-order the positions of the nodes of T. Each traversal can be performed in linear time. □

2.2. Dynamic environment. We consider a fully dynamic environment for the maintenance of $\square$-drawings on a collection of trees. Namely, we introduce the following set $\mathcal{O} = \mathcal{Q} \cup \mathcal{U}$ of operations:

- Query operations ($\mathcal{Q}$):
  - *Draw*(*node* v)—Return the (x, y) position of node v.
  - *Offset*(*node* v)—Return the (x, y) position of the reference point of $\square(v)$.
  - *DrawSubtree*(*node* v)—Return the subdrawing of the subtree rooted at node v.
  - *Window*(*node* v, *point* p, q)—Draw the portion of subtree T_v contained in the query window defined by lower-left corner p and upper-right corner q.
- Update operations ($\mathcal{U}$):
  - *MakeGraph*(*node* λ)—Create a new elementary tree T consisting of a single node.
  - *DeleteGraph*(*node* λ)—Remove the elementary tree consisting of the single node λ.
  - *Link*(*node* ρ, v, μ_1, μ_2)—Let ρ be the root of a tree and let v be a node of another tree. Also, let μ_1 and μ_2 be consecutive children of v. Add an edge from ρ to v and insert ρ between μ_1 and μ_2. If parameters μ_1 and μ_2 are not provided, then ρ becomes the only child of v. If $\mu_1(\mu_2)$ is not provided then ρ is inserted as the first (last) child of v.
  - *Cut*(*node* μ)—This operation assumes that μ is not the root of a tree. Remove the edge from μ to its parent, thus separating the subtree rooted at μ.
  - *Evert*(*node* μ)—Change the parent/child relationship for all nodes on the path from μ to the root of its tree, making μ the root. This operation maintains the clockwise order of neighbors of every node of T.
  - *Reflect*(*node* μ)—Reflect the subtree T_μ, i.e., reverse the order of the children of all the nodes of T_μ.
  - *Expand*(*node* v, μ', μ'')—Let $\mu_1, \ldots, \mu_k$ be the children of node v, and let $\mu' = \mu_i$ and $\mu'' = \mu_j (1 \leq i < j \leq k)$. Replace nodes $\mu_i, \ldots, \mu_j$ in the ordering of the children of v with a new node μ. Node μ has children $\mu_i, \ldots, \mu_j$.
  - *Contract*(*node* μ)—This is the inverse operation of *Expand*. It merges node μ with its parent.

In the rest of this section, we show how to dynamically maintain $\square$-drawings, and we prove the following theorem.

THEOREM 2.2. *Consider the following dynamic graph drawing problem:*

- *Class of graphs $\mathcal{G}$: forest of rooted ordered trees.*
- *Static drawing predicate $\mathcal{P}_S$: upward, planar, grid, straight-line, layered, order-preserving, centered, isomorphic, symmetric, quadratic-area drawing.*
- *Repertory of operations $\mathcal{O}$: Draw, Offset, DrawSubtree, Window, MakeGraph, DeleteGraph, Link, Cut, Evert, Reflect, Expand, and Contract.*
- *Dynamic drawing predicate $\mathcal{P}_D$: the drawing of a subtree not affected by an update operation changes only by a translation.*

There exists a fully dynamic algorithm for the above problem with the following performance:

- *A tree with n nodes uses $O(n)$ memory space;*
- *Operations MakeGraph and DeleteGraph each take $O(1)$ time;*
- *Operation DrawSubtree takes $O(\log n + k)$ time to return the position of k nodes and edges;*
- *Operation Window takes $O(k \cdot \log n)$ time to return the position of k nodes and edges;*
- *Operations Draw, Offset, Link, Cut, Evert, Reflect, Expand, and Contract each take $O(\log n)$ time.*

In §2.3 we present a data structure for maintaining an implicit representation of a $\square$-drawing. By Lemma 2.1, static predicate $\mathcal{P}_S$ is satisfied. In §2.4 we show how to perform query operations. In §2.5 we show how to perform update operations and discuss the dynamic drawing predicate $\mathcal{P}_D$.

2.3. Data structure. We recall that the y-axis is directed downward. Note that edges are still directed upward from each child to its parent. We use the following centering rule:

- $x(\mu) = \frac{1}{2}(x(\mu_\ell) + x(\mu_r))$,

where x_ℓ and x_r are the leftmost and rightmost descendants of μ, respectively.

2.3.1. Values to be maintained. In order to maintain the $\square$-drawing of a tree T we keep the following values for a node μ:

- *width*(μ)—The width of $\square(\mu)$.
- *level*(μ)—The level (distance from the root) of μ.
- *reference*(μ)—The x-coordinate of the reference point of $\square(\mu)$. (The y-coordinate of the reference point of $\square(\mu)$ is *level*(μ).)

Figure 2 shows the equations to calculate these values. Note that if μ is the root of T, then *reference*(μ) = 0. From these values we can easily compute the coordinates of a node μ. Namely, the x-coordinate of μ is *reference*(μ)+ *width*(μ)/2 and the y-coordinate of μ is *level*(μ).

LEMMA 2.3. *Consider a node σ and an ancestor τ of σ in tree T. Then the dependence of values width(τ), level(σ), and reference(σ) on values width(σ), level(τ), and reference(τ) can be expressed as*

$$\begin{aligned} \mathit{width}(\tau) &= \mathit{width}(\sigma) + a, \\ \mathit{level}(\sigma) &= \mathit{level}(\tau) + b, \\ \mathit{reference}(\sigma) &= \mathit{reference}(\tau) + c, \end{aligned}$$

where a, b, and c are independent from width(σ), level(τ), and reference(τ).

Proof. The proof is by induction on the length of the path from node σ to node τ. If $\sigma = \tau$ then the lemma is trivially true. The inductive step follows from the equations of Fig. 2. $\square$

Consider a directed path Π from node σ to an ancestor τ of σ in tree T. We refer to nodes σ and τ as the *head* and *tail* of path Π, respectively. Given such a σ, τ, and Π, the

$$
\begin{aligned}
width(\mu) &= \sum_{i=1}^{d} width(\mu_i), \\
level(\mu_i) &= level(\mu) + 1, \\
reference(\mu_i) &= reference(\mu) + \sum_{j=1}^{i-1} width(\mu_j).
\end{aligned}
$$

FIG. 2. *The equations to calculate the values of width for a node μ and the values of reference and level for the children $\mu_1, \ldots, \mu_d$ of μ.*

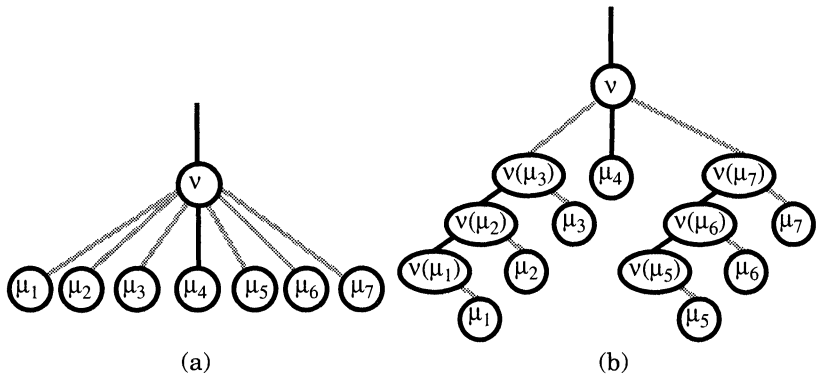


FIG. 3. *The edge-paths for a node v with respect to a solid path.* (a) *Node v and its children.* (b) *The left and right edge-paths.*

transfer vector $h(\Pi)$ is the 3-tuple (a, b, c) of values for the calculation of $width(\tau)$, $level(\sigma)$, and $reference(\sigma)$ (see Lemma 2.3).

2.3.2. A tree representation technique. Our data structure consists of representing an n-node tree T as a collection of disjoint directed paths, where edges are directed from child to parent. We partition the edges of T into *solid* or *dashed* such that at most one solid edge is incoming into a node. Therefore, every nodes is in exactly one maximal path of solid edges (of length 0 or more), called a *solid path* of T.

The partition of edges into solid and dashed is obtained by means of the following *size invariant*: let $size(\mu)$ denote the number of nodes in the subtree of T rooted at node μ. An edge (μ, v) of T is solid if $size(\mu) > size(v)/2$. If the partition satisfies the size invariant, then every path of T has $O(\log n)$ dashed edges.

Consider a tree T containing a node v with children $\mu_1, \ldots, \mu_d$ in left-to-right order. Suppose Π is a path T containing both v and a child μ_i. We associate with v two paths, $\Pi_\ell(v)$ and $\Pi_r(v)$, called the *left edge-path* and *right edge-path* of v with respect to Π (see Fig. 3). Left edge-path $\Pi_\ell(v)$ consists of a node $v(\mu_j)$ for each child μ_j of v, $1 \leq j < i$, and edges directed from $v(\mu_j)$ to $v(\mu_{j+1})$. Similarly, right edge-path $\Pi_r(v)$ is formed from children μ_j of v, $i < j \leq d$.

In our data structure, we represent as balanced search trees the following paths:

- the solid paths of T; and
- for each node v of T, the left and right edge-paths of v with respect to the solid path through v.

For each such path Π, the associated balanced binary tree is denoted B_Π and is called the *path-tree* of Π. Also, we identify Π with the root ζ of B_Π. Each leaf λ of B_Π represents a node of Π, and the left-to-right order of the leaves of B_Π corresponds to the head-to-tail order of the nodes of Π.

Suppose Π is a solid path. Each leaf λ of B_Π represents a node μ of Π. Each internal node η of B_Π represents the subpath $\Pi(\eta)$ of Π associated with the leaves in the subtree of η. Note that we identify path $\Pi(\eta)$ with η. We store at node η four versions of the transfer vector: $h(\eta), \overline{h}(\eta), h^*(\eta), \overline{h^*}(\eta)$, corresponding to no restructuring of T, reversing the parent–child relationship for nodes of Π, reflecting the subtrees rooted at nodes of Π, and both reversing the parent–child relationship and reflecting the subtrees rooted at nodes of Π. If Π is an edge-path we keep the value *width*(Π) corresponding to the sum of the widths of the bounding boxes of the tree nodes associated with the nodes of $\Pi(\eta)$. We also keep the *value invariant* that corresponds to the tail τ of every solid path stores the actual value of *width*(τ).

In order to achieve the logarithmic time per dynamic operation, we use biased search trees [1] to represent the path-trees. Each leaf of a biased search tree is assigned a positive weight. Biased search trees use linear space and have the property that, if w is the weight of leaf λ and W is the sum of the weights of all the leaves, the depth of λ is $O(\log(W/w))$.

For a node v of a solid path Π, we define *weight*(v) as one plus the sum of the sizes of the subtrees connected to v by dashed edges. Similarly, for a node $v(\mu)$ of edge-path $\Pi_X(v)(X = \ell$ or $X = r)$, *weight*$(v(\mu))$ is defined to be one plus the total weight of the solid path containing node μ. For each leaf of a path-tree we define its weight equal to the weight of the corresponding path node. Each path-tree is then implemented as a biased search tree. Note that this is consistent with the definition of biased search trees [1], since the weights of the leaves are positive.

Consider a node μ of T on solid path Π. We keep a boolean value *reversed*(μ) indicating which of the neighbors of μ on Π is its parent. We also keep a boolean value *reflected*(μ) which indicates the *path reflection status*, which is the left-to-right direction of the children of μ, ignoring any reflection done by ancestors of *tail*(Π). At each path-tree node η, we keep the associated offset values *reversed*(η) and *reflected*(η). These offset values are kept such that for any node μ, the true value of *reflected*(μ) or *reversed*(μ) is the exclusive-or of the values stored at the path-tree nodes on the path from μ to the root of the path-tree containing μ.

At each node μ of a path Π, we store pointers *left*(μ) and *right*(μ) to the left and right neighbors of μ on Π. At each path-tree node η we store pointers to *head*$(\Pi(\eta))$ and *tail*$(\Pi(\eta))$. If *reversed*$(\mu) = 1$, then *left*(μ) points to the right neighbor of μ and *right*(μ) points to the left neighbor of μ. If *reversed*$(\eta) = 1$, then *head*(η) points to *tail*$(\Pi(\eta))$ and *tail*(η) points to *head*$(\Pi(\eta))$. Also, *reversed*$(\eta) = 1$ swaps the meaning of the left and right edge-path pointers in the subtree of η. The values of the *reversed* and *reflected* bits determine the actual meaning of the four values kept for transfer vector h at η.

Since the path-trees have a total of $O(n)$ nodes and each node of the tree and of the path-trees uses $O(1)$ memory space, we conclude that the data structure uses $O(n)$ memory space.

2.3.3. Operation *push*. The following auxiliary operation (to be used in the next sections) moves the values of *reflected* and *reversed* toward the leaves of a path-tree in $O(1)$ time:

• *push*(*node* η)—For internal path-tree node η with children η' and η'', combine the values *reversed* and *reflected* at nodes η' and η'' with those at η, and set *reversed*$(\eta) =$ *reflected*$(\eta) = 0$.

Operation *push*(η) on internal path-tree node η is implemented as follows:
(1) If η is an internal node, we set

$$\begin{aligned} \textit{reflected}(\eta') &= \textit{reflected}(\eta') \oplus \textit{reflected}(\eta);\\ \textit{reflected}(\eta'') &= \textit{reflected}(\eta'') \oplus \textit{reflected}(\eta);\\ \textit{reversed}(\eta') &= \textit{reversed}(\eta') \oplus \textit{reversed}(\eta);\\ \textit{reversed}(\eta'') &= \textit{reversed}(\eta'') \oplus \textit{reversed}(\eta), \end{aligned}$$

where operation $\oplus$ is the bit-or operation. Also, if $\textit{reversed}(\eta) = 1$, we exchange pointers $\textit{head}(\eta)$ and $\textit{tail}(\eta)$, values $h(\eta)$ and $\overline{h}(\eta)$, and values $h^*(\eta)$ and $\overline{h^*}(\eta)$. If $\textit{reflected}(\eta) = 1$, we exchange values $h(\eta)$ and $h^*(\eta)$, and values $\overline{h}(\eta)$ and $\overline{h^*}(\eta)$.

(2) Otherwise, η is a leaf of some path-tree associated with some (edge or solid) path Π. Therefore, η is a node of Π. Operation *push* maintains the effect of the values *reversed* and *reflected* for η and its descendants in the following manner. If $\textit{reversed}(\eta) = 1$, then we exchange pointers $\textit{left}(\eta)$ and $\textit{right}(\eta)$. If Π is a solid path, we also exchange pointers to the left and right edge-paths. If $\textit{reflected}(\eta) = 1$, then we flip the *reflected* bits at the root of any path-tree pointed to by node η. If Π is a solid path, we also flip the *reversed* bits at the root of the path-trees representing the left and right edge-paths of η.
(3) We set $\textit{reversed}(\eta) = \textit{reflected}(\eta) = 0$.

2.4. Query operations. The query operations *Locate*, *Offset*, *DrawSubtree*, and *Window* are implemented using the following suboperations derived from dynamic trees [36], each of which takes $O(\log n)$ time to perform:

• *splice*(*path* Π)—This operation assumes that Π is a solid path ending at $\mu \neq \rho$. Convert the dashed edge leaving μ to solid and convert the solid edge (if it exists) entering the parent ν of μ to dashed.

Suppose node μ' is a sibling of μ such that there is a solid edge from μ' to ν. Let Π' be the solid path containing nodes μ' and ν. Let $\Pi_\ell(\nu)$ and $\Pi_r(\nu)$ be the left and right edge-paths of ν with respect to Π'. We convert solid edge (μ', ν) to dashed as follows. We begin by splitting path Π' at ν. We then create a path-tree node $\nu(\mu')$ for the new dashed edge and set the left edge-path of ν to be the concatenation $\Pi_\ell(\nu)$, $\nu(\mu)$, and $\Pi_r(\nu)$. The right edge-path becomes the empty path.

Converting the dashed edge from μ to ν to solid is performed similarly, reversing the roles of the solid and edge-paths. Let Π'' be the path with $\nu = \textit{head}(\Pi'')$. Split $\Pi_\ell(\nu)$ at $\nu(\mu)$. The resulting paths to the left and right of $\nu(\mu)$ become the left and right edge-paths of ν. We then concatenate Π and Π'' to create the solid edge.

• *expose*(μ)—Convert to dashed the solid edge entering μ, if such an edge exists. Create a solid path from μ to the root by converting to solid all the dashed edges (ν', ν'') of such a path, and converting to dashed the edges $(\textit{sib}(\nu'), \nu'')$. Operation *expose*($\lambda$) consists of a sequence of *splice* operations on the solid paths containing the nodes on the path from λ to ρ. This operation is always followed by a *conceal* operation, (see below) which undoes its effect.

• *conceal*(μ)—Restore the original type (solid or dashed) of the edges entering the nodes on the path Π from node μ to the root ρ. This operation is the inverse of *expose* and also consists of a sequence of *splice* operations [36].

Additionally, we implement operation *Window* using the following auxiliary operations:

• *locatepoint*(*node* ν; *point* p)—Return the node μ of T_ν such that p is contained in $\square(\mu)$, but p is not contained in the bounding rectangle of any of the children of μ. If p is outside $\square(\nu)$, then *locatepoint*(ν, p) returns *nil*.

• *drawproper*(*node* v; *point* p, q)—Draw node v and the edges from v to its children, clipping to window W defined by lower-left corner p and upper-right corner q.

• *findrightint*(*node* v; *point* p, q) (*findleftint*(*node* v; *point* p, q))—Let Π be the path following right (left) children from v. Return the closest descendant μ of v on Π such that *drawproper*(μ, p, q) draws (at least) a portion of an edge.

• *drawtree*(*node* v, *point* p, q)—Draw the portion of subtree G_v contained in the query window W defined by lower-left corner p and upper-right corner q. (Note that this operation is mutually recursive with operation *Window*.)

Operations *locatepoint*, *drawproper*, *findrightint*, and *findleftint* take $O(\log n)$ time each to perform.

Operation *Draw*(v) is performed by first issuing *expose*(v) to get path Π. By the value invariant, node v will store the value of *width*(v). The transfer vector for Π contains the value of *level*(v) and *reference*(v). Therefore, the y-coordinate returned is *level*(v) and the x-coordinate is *reference*(v)+(*width*(v)/2). Operation *Offset*(v) is realized similarly: we first call *expose*(v). The x-coordinate of the reference point of $\square(v)$ is *reference*(v) and the y value is *level*(v). Operation *DrawSubtree*(v) is performed by calling *Offset*(v) in order to determine the position of $\square(v)$. We then use the sequential algorithm to draw T_v.

We implement operation *Window*(v, p, q) by performing the following steps:

(1) If W does not intersect $\square(v)$ then stop.

(2) Clip W to $\square(v)$, update points p and q accordingly.

(3) Initialize scan point $s = (x_s, p_s)$ to be point p. Let node v = *locatepoint*(v, s). We call *drawtree*(μ, s, q).

(4) Move scan point s by resetting x_s to be 0.25 plus the x-coordinate of the right edge of $\square(\mu)$. Repeat steps 3 and 4 until s is no longer contained in $\square(v)$ or W.

Operation *drawtree*(v, p, q) is implemented by performing the following steps:

(1) Perform *drawproper*(v, p, q).

(2) If node v is a leaf, or if $y_q < depth(v) + 1$ then stop.

(3) If step 1 did not draw any edges, then suppose *Draw*(v) is to the left (right) of W. Let μ = *findrightint*(v, p, q)(μ = *findleftint*(v, p, q)). If $\mu = nil$ then stop. Otherwise call *drawtree*(μ, p, q).

(4) If step 3 did draw edges then let $y_p = depth(v) + 1$ and call *Window*(μ, p, q).

Each step is implemented in $O(\log n)$ time for each edge and node drawn. After drawing an edge, we take $O(\log n)$ time to determine if we stop. Therefore, to draw k nodes and edges takes $O(k \cdot \log n)$ time.

2.4.1. Extensions to dynamic trees. In this and the next section, we describe the auxiliary operations used in the implementation of the query operations. We first describe the extensions to dynamic trees required for operations *splice*, *expose*, and *conceal*. As described above, operations *expose* and *conceal* are implemented as repeated calls to operation *splice*. Operation *splice* concatenates and splits balanced binary trees representing solid paths. We use the following elementary tree operations, each taking $O(1)$ time:

• *join*(*node* ζ', ζ'')—Given the roots ζ' and ζ'' of two binary trees representing solid paths Π' and Π'', combine the trees into a new tree by creating a new root ζ with left child ζ' and right child ζ''. Let σ' and τ' be the head and tail of Π' and σ'' and τ'' be the head and tail of Π''. If paths Π' and Π'' are edge-paths, then we have *weight*(Π) = *weight*(Π')+*weight*(Π'').

Now, suppose Π' and Π'' are solid paths. We show how to calculate transfer vector $h(\Pi)$ in $O(1)$ time. The calculations for transfer vectors $\overline{h}(\Pi)$, $h^*(\Pi)$, and $\overline{h^*}(\Pi)$ are performed similarly.

Suppose $h(\Pi) = (a, b, c)$, $h(\Pi') = (a', b', c')$, and $h(\Pi'') = (a'', b'', c'')$ are the transfer vectors for paths Π, Π' and Π''. Suppose w_ℓ and w_r are the total width of the left and right

edge-paths of node σ''. We calculate $h(\Pi)$ in the following manner:

$$\begin{aligned} width(\tau'') &= width(\sigma'') + a'', \\ width(\tau'') &= (width(\tau') + w_\ell + w_r) + a'', \\ width(\tau'') &= ((width(\sigma') + a') + w_\ell + w_r) + a'', \end{aligned}$$

$$\begin{aligned} level(\sigma') &= level(\tau') + b', \\ level(\sigma') &= (level(\sigma'') + 1) + b', \\ level(\sigma') &= ((level(\tau'') + b'') + 1) + b', \end{aligned}$$

$$\begin{aligned} reference(\sigma') &= reference(\tau') + c', \\ reference(\sigma') &= (reference(\sigma'') + w_\ell) + b', \\ reference(\sigma') &= ((reference(\tau'') + c'') + w_\ell) + c'. \end{aligned}$$

Therefore, we have

$$\begin{aligned} a &= a' + a'' + w_\ell + w_r, \\ b &= b' + b'' + 1, \\ c &= c' + c'' + w_\ell. \end{aligned}$$

• *separate*(*node* ζ)—Given the root ζ of a binary tree, divide the tree into two trees with roots ζ' and ζ'', where ζ' is the root of the left subtree and ζ'' is the root of the right subtree. The transfer vectors of ζ' and ζ'' do not change. To do this, we call *push*(ζ) to preserve the effects of *reversed*(ζ) and *reflected*(ζ).

• *rotateleft*(*node* v) (*rotateright*(*node* v))—Perform a left (right) rotation at node η. This operation can be performed with $O(1)$ *separate* and *join* operations.

Since path-trees are biased by the weights, the sum of the number of elementary tree operations at each *splice* operation telescopes, so that operations *expose* and *conceal* can be performed with $O(\log n)$ elementary tree operations.

2.4.2. Operations using selection functions. In order to implement auxiliary operations *locatepoint*, *drawproper*, *findrightint*, and *findleftint*, we introduce selection functions, which are used to find distinguished nodes of a path or tree. We will also use this structure in the implementation of operations in §3.

A *path-selection function* S maps a path Π and a query argument q into a node $\mu = S(\Pi, q)$ of Π, such that if Π is the concatenation of Π' and Π'', then one can determine in $O(1)$ time whether μ is in Π' or Π'' from q and the transfer values stored for Π.

A *tree-selection function* S maps a tree T and a query argument q into a node $\mu = S(T, q)$ of T. Function S is a path-selection function. If node μ is a descendant of the tail of path Π of tree T, then *find*(Π, S, q) returns the deepest node μ' on Π such that μ is a descendant of μ'.

We then use the following operations to find the distinguished nodes:

• *PathFind*(*node* μ', μ''; *selectionfunction* S)—Let Π be the path of a tree from node μ' to node μ''. Find the node of Π returned by the path-selection function S.

• *TreeFind*(*node* v; *selectionfunction* S)—Find the node of the subtree rooted at μ returned by the tree-selection function S.

Operations *PathFind* and *TreeFind* use the following function which returns the distinguished node of a path:

• *find*(*path* Π, *selectionfunction* S, *value* q)—Find node μ of Π returned by path-selection function S with query argument q.

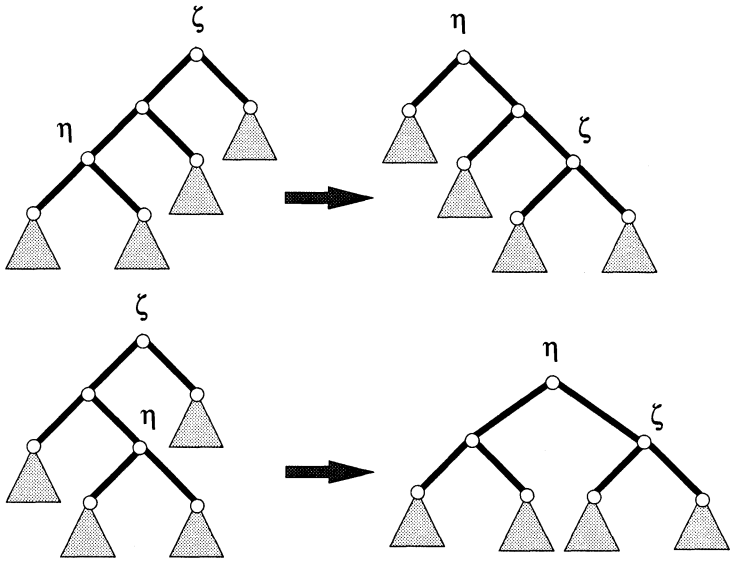


FIG. 4. *The rotations performed in operation splaystep*(η).

We need the following operation from [37] in our implementation of *find*(Π, S, q):

• *splaystep*(*node* η)—For binary tree B with root ζ and node η a grandchild of ζ, restructure B such that the relative order of the leaves of B remain fixed and every node in the subtree rooted at η has its depth reduced by one or two. We accomplish this as follows. Let η' be the child of ζ that is the parent of η. If η' and η are both left children, then perform *rotateright*(ζ) twice. If η' is a left child and η is a right child, then perform *rotateleft*(η') followed by *rotateright*(ζ). The other two cases are symmetric (see Fig. 4).

Operation *splaystep* is implemented with a constant number of *rotateleft* and *rotateright* operations. Hence, *splaystep* takes $O(1)$ time.

LEMMA 2.4. *Suppose S is a path-selection function and ζ is the root of a pathtree Π. Then given query argument q, we can determine in $O(1)$ time if $S(\Pi, q)$ is a child or grandchild of ζ, or which grandchild of ζ is the root of the subtree containing $S(\Pi, \zeta)$.*

Proof. Suppose η_1 and η_2 are the children of ζ. By definition, in $O(1)$ time we can determine which subtree rooted at η_1 or η_2 contains $S(\Pi, q)$. If the subtree found is a single node, then it is $S(\Pi, q)$. Without loss of generality, suppose we know that $S(\Pi, q)$ is in the subtree rooted at η_1 and η_1 is not a leaf. Then let η_{11} and η_{12} be the children of η_1. We then perform $O(1)$ rotations on B_Π such that η_{11} becomes the left child of the root. Then, in $O(1)$ time we can determine if $S(\Pi, q)$ is in the subtree rooted at η_{11}. Otherwise, $S(\Pi, q)$ is in the subtree rooted at η_{12}. □

Operation *find*(ζ) is implemented as follows, starting at node ζ and repeating until a value is returned: if μ is a child or grandchild of ζ then return μ. Otherwise, determine the subtree rooted at the grandchild η of ζ which contains μ. Do *splaystep* at η and recur. After μ is found, we undo all the *splaysteps* to restore the balance of the path-tree. This takes $O(d_\mu)$ time where d_μ is the depth of the returned node μ.

Operation *PathFind*(μ', μ'', S, q) makes two calls to *expose* to get the path Π from μ' to μ'', then calls *find*(Π, S, q). We then restore the weight invariant by calling *expose*(v') followed by *conceal*(v').

Consider tree selection function S. We implement operation *TreeFind*(v, S, q) as follows. If v is not the root of its tree, we first call *expose* at *Parent*(v). Node v then becomes the tail

of its solid path Π. Let μ be the node we are searching for. We repeat the following until node μ is found. Let node $\mu' = \mathit{find}(\Pi, S_p, q)$. Create path Π' with head μ' and tail v. This is done with single *split* and *concatenate* operations.

Assume node μ' is a tree node. Let path Π'' be the result of joining the left edge-path $\Pi_\ell(\mu')$ and Π'. This is well defined by the expansion invariant. By the definition of tree and path selection functions, we can determine in $O(1)$ time if node μ is in $\Pi_\ell(\mu')$. If not, repeat using the right edge-path $\Pi_r(\mu')$.

If neither edge-path has μ as a descendant, then $\mu' = \mu$. Otherwise, we reset path Π to be Π'' and continue. If μ' is an edge-path node, we proceed similarly, except we join the solid-path associated with μ' to Π'. After μ is found, we undo the restructuring to restore the path-trees.

At each iteration, the time to find μ' and create Π'' is $O(d_{\mu'})$, where $d_{\mu'}$ is the depth of μ' in B_Π, the path-tree for path Π. Operation *join* increases the depth of a path-tree node by 1. Therefore, $d_{\mu'}$ is at most one more than the depth of μ' in its original path-tree. Therefore, the time to find μ is $O(\log n)$ plus the time to perform $\mathit{expose}(\mu)$. The time to restore the path-trees is equivalent. Therefore operation *TreeFind* is implemented in $O(\log n)$ time.

Suppose $p = (x_p, y_p)$ and $q = (x_q, y_q)$ are points with $x_p < x_q$ and $y_p < y_q$, and let W be the window defined by p and q. Recall that the y-coordinate of the drawing of a node μ of tree T corresponds to the depth of μ in T.

We use the following fact in our algorithm. If x_ℓ is the x-coordinate of the reference point of the bounding box of some node μ, then the line $x = x_\ell + 0.25$ does not interest any edge or node in the drawing of T_μ.

We keep the following additional value for each solid path Π in T:

- $\mathit{rightmost}(\Pi)$—**true** if and only if each node of Π other than $\mathit{tail}(\Pi)$ has no right sibling.
- $\mathit{leftmost}(\Pi)$—**true** if and only if each node of Π other than $\mathit{tail}(\Pi)$ has no left sibling.

Clearly we can maintain the value *rightmost* during join operations.

Operation $\mathit{locatepoint}(v, p)$ first calls $\mathit{expose}(v)$ in order to find the location and width of $\square(v)$. If point p is outside $\square(v)$, then return nil. Otherwise, we use the following tree selection function, which takes query point p as an argument.

Selection function S_1. *Suppose η is the root of a path-tree with children η' and η''. If point p is contained in $\square(\mathit{tail}(\eta'))$, then return η', otherwise return η''.*

Operation $\mathit{TreeFind}(v, S_1, p)$ returns a node μ such that p is contained in $\square(\mu)$, but p is not contained in the bounding box for any of the children of μ. We perform operation *locatepoint* by setting node $\mu = \mathit{TreeFind}(v, S_1, p)$. If p is on the right boundary of $\square(\mu)$, then let $x_p = x_p + 0.25$ and repeat. We then return node μ.

We perform operation $\mathit{drawproper}(\mu, p, q)$ by first calling $\mathit{expose}(\mu)$. The following path selection function on edge-path $\Pi_\ell(\mu)$ takes as an argument the pair of points (p, q) that define window W.

Selection function S_2. *Suppose η is the root of a path-tree with children η' and η''. Let μ' be the child of μ associated with $\mathit{tail}(\eta')$. If the edge from node μ' to node μ intersects W, then return η', otherwise return η''.*

We find the leftmost edge of the proper region of μ intersecting W by letting node $\mu_\ell = \mathit{PathFind}(\Pi_\ell(\mu), S_2, (p, q))$. We find the rightmost intersecting edge connected to node μ_r similarly. We then draw μ, and the edges between μ_ℓ and μ_r, clipping to W.

Operation *findrightint* is performed using the following tree selection function which takes an argument x_ℓ, the x-coordinate of the left side of the query window.

Selection function S_3. *Suppose η is the root of a path-tree with children η' and η''. Construct node η'' representing the path from $\mathit{tail}(\eta')$ to $\mathit{tail}(\eta'')$. If $\mathit{rightmost}(\eta''') =$* **false** *of if the x-coordinate of $\mathit{tail}(\eta')$ is greater than x_ℓ, then return η''. Otherwise return η'.*

When used with operation *TreeFind*, selection function S_3 returns the first descendant μ' of μ reached through only rightmost children such that there is an edge from μ' which intersects the line $x = x_\ell$. Operation *findrightint*(v, p, q) is then implemented as follows. Let node $\mu = \mathit{TreeFind}(v, S_3, x_p)$. If μ is a leaf then return *nil*. Otherwise, let node μ' be the rightmost child of μ. If the edge from node μ' to node μ intersects W, return μ. Otherwise return *nil*. Operation *findleftint*(v, p, q) is implemented in a similar manner, substituting the value *leftmost* for *rightmost*.

2.5. Update operations. Operations *MakeGraph* and *DeleteGraph* can be trivially implemented in $O(1)$ time. Operations *Link* and *Cut* can each be implemented as variations of the dynamic tree operations *Link* and *Cut*.

Operation *Evert*(μ) consists of issuing *expose*(μ), flipping the reverse bit of the resulting solid path, then restoring the size-invariant by calling *conceal*(μ). This is implemented in $O(\log n)$ time.

Operation *Reflect*(μ) is performed as follows. First, we call *expose*(μ). The left edge-path of μ then contains nodes for all the children of μ. We reverse the order of the children of μ by flipping the *reflected* and *reversed* bits at the root of the path-tree representing $\Pi_\ell(\mu)$. We flip the reflected bit to reflect all subtrees rooted at children of μ and the *reversed* bit to reverse the order of the children of μ. Finally, we perform *conceal*(μ) to restore the original solid and dashed edges. Hence, operation *Reflect* is implemented in $O(\log n)$ time.

Operation *Expand*(μ, μ', μ'') begins by calling *expose*(μ) which results in the left edge-path of μ, $\Pi_\ell(\mu)$ containing all the edges from children to μ. Then we perform a constant number of split and join operations to form three subpaths Π', Π'', and Π''' of $\Pi_\ell(\mu)$, with Π' containing the edges preceding edge (μ', μ), Π'' containing the edges between edges (μ', μ) and (μ'', μ), and Π''' containing the edges following edge (μ'', μ). Paths Π' and Π''' become the left and right edge-paths of μ. We create a new node v with left edge-path Π'' and right edge-path the empty path. We extend the solid path containing μ by concatenating a new node v. Finally, we call *conceal*(v) to restore the size invariant. Hence, operation *Expand* takes $O(\log n)$ time.

Operation *Contract*(μ) begins by calling *expose*(μ) which results in the left edge-path of μ, $\Pi_\ell(\mu)$, containing all the edges from children to μ. Let node v be the parent of μ. Next, we remove node μ. We set $\Pi_\ell(v)$ to the concatenation of $\Pi_\ell(v)$, $\Pi_\ell(\mu)$, and $\Pi_r(v)$. We set $\Pi_r(v)$ to the empty path. Finally, we call *conceal*(v) to restore the size invariant. Hence, operation *Contract* takes $O(\log n)$ time.

In a $\square$-drawing, the drawing of a subtree is independent, up to a translation, from the structure of the rest of the tree. Hence, dynamic predicate $\mathcal{P}_D$ is satisfied.

3. Series parallel digraphs. A *source* of a digraph is a vertex without incoming edges. A *sink* is a vertex without outgoing edges. A *pole* is either a source or a sink. A series-parallel digraph is a simple digraph with exactly one source s and one sink t, recursively defined as follows:

- A digraph consisting of a single edge from s to t is a series-parallel digraph.
- Given series-parallel digraphs $G_1, \ldots, G_k$ with sources $s_1, \ldots, s_k$ and sinks $t_1, \ldots, t_k$, the digraph obtained by identifying sink t_i with source s_{i+1} for $1 \leq i < k$ is a series-parallel digraph, with source s_1 and sink t_k. Vertices $v_i = t_i = s_{i+1}$, $1 \leq i < k$ are called the *join-vertices* of such a composition. This is called *series composition.*
- Given series-parallel digraphs $G_1, \ldots, G_k$, with sources $s_1, \ldots, s_k$ and sinks $t_1, \ldots, t_k$, the digraph obtained by identifying $s_1, \ldots, s_k$ into a single vertex s and identifying $t_1, \ldots, t_k$ into a single vertex t is a series-parallel digraph. This is called *parallel composition.* Since multiple edges are not allowed, at most one of the G_i's consists of a single edge.

It is well known that an n-vertex series-parallel digraph is planar and has $O(n)$ edges. A series-parallel digraph G is associated with a rooted tree T, called the *SPQ-tree*. A node $\mu \in T$ represents a subgraph of G, called the *pertinent digraph* of μ, and denoted by G_μ. If μ is a leaf, then G_μ is a single edge. Otherwise, G_μ is obtained by the composition (series or parallel) of the pertinent digraphs of the children of μ. The nodes of T are of four types: S-nodes, P-nodes, P_Q-nodes, and Q-nodes. Tree T is defined recursively as follows:

- If G is a single edge, then T consists of a single *Q-node*.
- If G is the parallel composition of series-parallel digraphs $G_1, \ldots, G_k$ with SPQ-trees $T_1, \ldots, T_k$ with roots $\rho_1, \ldots, \rho_k$, and none of the G_i, $1 \leq i \leq k$ is a single edge, then T consists of a *P-node* root with children $\rho_1, \ldots, \rho_k$.
- If G is the parallel composition of series-parallel digraphs G_1 and G_2 with SPQ-trees T_1 and T_2 with roots ρ_1 and ρ_2, and G_2 is a single edge, then T consists of a *P_Q-node* root with children ρ_1 and ρ_2.
- If G is the series composition of series-parallel digraphs $G_1, \ldots, G_k$ with SPQ-trees $T_1, \ldots, T_k$ with roots $\rho_1, \ldots, \rho_k$, then T consists of an *S-node* root with children $\rho_1, \ldots, \rho_k$.

We keep two types of nodes representing parallel composition since transitive edges are handled differently from general parallel composition.

The *type* of a node v is either Q, P, P_Q, or S. We keep the type invariant that each node of an SPQ-tree T does not have a child of the same type, and that a P-node cannot have P_Q-node as a child. If G has n vertices, then T has $O(n)$ nodes. Tree T can be constructed in $O(n)$ time using the recognition algorithm of [49].

We can represent an embedding of a series-parallel digraph by the ordering of the children of the P- and P_Q-nodes of the SPQ-tree. A *right-pushed* embedding is such that all the transitive edges are embedded on one side, say, the right side. Right-pushed embeddings are used in the drawing algorithm of §3.1.

Let v be an S-node with children $\mu_1, \ldots, \mu_k$. The *skeleton* of v, denoted *skeleton*(v), is the series-parallel digraph consisting of k edges $e_i = (v_{i-1}, v_i)$, $1 \leq i \leq k$, where v_0 and v_k are the source and sink of the pertinent digraph of v, and for $1 \leq i < k$, v_i is the sink of the pertinent digraph of μ_i. The *proper node* of vertices $v_1, \ldots, v_{k-1}$ is node v. Note that v_i is a join-vertex used in the series composition at its proper node. Hence, if G is a series-parallel digraph with associated SPQ-tree T then each vertex of G, with the exception of its poles, has a proper node.

Given a digraph G, the *reverse* digraph $\bar{G}$ of G is formed by reversing the orientation of all edges of G. It is easy to see that if G is a series-parallel digraph, then $\bar{G}$ is also a series-parallel digraph. If s and t are the source and sink of G, respectively, then t and s are the source and sink of $\bar{G}$, respectively. If G is the parallel composition of series-parallel digraphs $G_1, \ldots, G_k$, then $\bar{G}$ is the parallel composition of series-parallel digraphs $\overline{G_1}, \ldots, \overline{G_k}$. Similarly, if G is the series composition of series-parallel digraphs $G_1, \ldots, G_k$, then $\bar{G}$ is the series composition of series-parallel digraphs $\overline{G_k}, \ldots, \overline{G_1}$.

Suppose G is a series-parallel digraph and T is its associated SPQ-tree. Let μ be a node of T and $\mu_1, \ldots, \mu_k$ be the children of μ. A *closed component* of G is either G or the union of the pertinent digraphs of a subsequence $\mu_i, \ldots, \mu_j$, where $1 < i \leq j < k$ and μ is an S-node. An *open component* of G is the union of the pertinent digraphs of a subsequence $\mu_i, \ldots, \mu_j$, minus its poles, where $1 \leq i \leq j \leq k$. A *component* is either an open or a closed component.

3.1. Δ-drawings. In this section, to simplify formulas we assume that the x-axis is directed from right to left and the y-axis is directed from bottom to top (see Fig. 5).

We consider the following static drawing predicate $\mathcal{P}_S$:

Upward: The drawing is upward.

Planar: The drawing is planar.

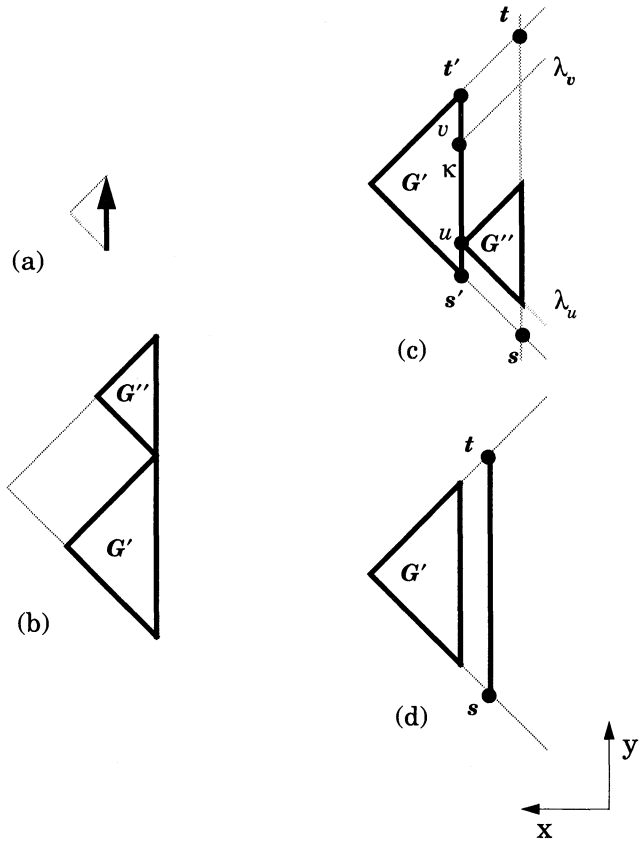


FIG. 5. *Geometric construction of a Δ-drawing:* (a) *base case;* (b) *series composition;* (c) *parallel composition (general case);* (d) *parallel composition with the "right-pushed" transitive edge.*

Grid: Vertices are placed at integer coordinates.

Straight-line: Edges are drawing as straight line segments.

Quasi-embedding-preserving: The drawing preserves the embedding, except, possibly, for the transitive edges.

Isomorphic: Isomorphic components have drawings that are congruent up to a translation.

Vertically symmetric: The drawings of a series-parallel digraph and its reverse have drawings that are congruent up to a translation and a reflection.

Quadratic-area: The drawing has $O(n^2)$ area.

It is important to note that in order to get polynomial area the embedding cannot be completely preserved. Namely, it is shown in [2] that there exists a class of embedded series-parallel digraphs for which any upward straight-line drawing that preserves the embedding requires exponential area under any resolution rule.

Δ-drawings of series-parallel digraphs are introduced in [2] and satisfy the above static drawing predicate. In Δ-drawings the embedding is modified into a right-pushed embedding (i.e., with all the transitive edges embedded on the right side). The Δ-drawing Γ of a series-parallel digraph G is inductively defined inside a *bounding triangle* $\Delta(\Gamma)$ that is isosceles and right angled. The hypotenuse of $\Delta(\Gamma)$, from now on called the *right side* of $\Delta(\Gamma)$, is a vertical segment, and the other two sides are on its left. The height of $\Delta(\Gamma)$ is the length of the right side; the width of $\Delta(\Gamma)$ one half of the height. In a series composition, the subdrawings are

placed one above the other. In a parallel composition, the subdrawings are placed one to the right of the other and are deformed in order to identify the poles, guaranteeing that their edges do not cross. The algorithm is outlined below. More details can be found in [2].

- Modify the embedding of G into a right-pushed embedding.
- If G consists of a single edge, it is drawn as a vertical segment of height 2, with bounding triangle having width 1 (see Fig. 5(a)).
- If G is the series composition of G' and G'', the drawings Γ' and Γ'' of G' and G'' are recursively constructed and combined by translating Γ'' so that its source is identified with the sink of G' (see Fig. 5(b)). The bounding triangle $\Delta(\Gamma)$ is obtained by extending the bottom side of $\Delta(\Gamma')$ and the top side of $\Delta(\Gamma'')$.
- If G is the parallel composition of G' and G'', the drawings Γ' and Γ'' of G' and G'' are recursively constructed. We consider the rightmost outgoing edge (s', u) of the source s' and G' and the rightmost incoming edge (v, t') of the sink t' of G' (see Fig. 5(c)–(d)). Let λ_u be the line through u that is parallel to the bottom side of $\Delta(\Gamma')$ and λ_v be the line through v that is parallel to the top side of $\Delta(\Gamma')$. Also, let κ be the vertical line extending the right side of $\Delta(\Gamma')$. We call the (infinite) region to the right of κ, λ_u, and λ_v the *prescribed region* of Γ''. First, we translate Γ'' anywhere inside its prescribed region. Then we identify the sources and sinks of G' and G'' by moving them to the intersections s and t of the right side of Γ'' with the lines extending the bottom and top sides of Γ'.

If the series or the parallel compositions involve more than two graphs (say ℓ graphs), the above steps are applied $\ell - 1$ times.

In a Δ-drawing Γ the source (resp., sink) of G is placed at the bottom (resp., top) vertex of $\Delta(\Gamma)$, and the other vertex of $\Delta(\Gamma)$ is not occupied by any vertex of G. Also, the rightmost outgoing edge of the source and the rightmost incoming edge of the sink lie on the right side of $\Delta(\Gamma)$.

We obtain an $O(n^2)$-area Δ-drawing by specializing the placement of Γ'' in a parallel composition so that $\Delta(\Gamma'')$ touches $\Delta(\Gamma')$. Let p be the point on the right side of $\Delta(\Gamma')$ and halfway between u and v. We translate Γ'' so that the left vertex of $\Delta(\Gamma'')$ coincides with p. In this way, Γ'' is in its prescribed region.

Notice that in both series and parallel compositions the height of $\Delta(\Gamma)$ is equal to the sum of the heights of $\Delta(\Gamma')$ and $\Delta(\Gamma'')$. Hence, the height of $\Delta(\Gamma)$ is exactly twice the number of edges, and the area of the drawing Γ is $O(n^2)$.

LEMMA 3.1 [2]. *Given an n-vertex series-parallel digraph G, the Δ-drawing of G satisfies the static drawing predicate $\mathcal{P}_S$, and can be constructed in $O(n)$ time.*

3.2. Dynamic environment. For a node μ of SPQ-tree T, we define $\Delta(\mu)$ as the *bounding triangle* to be the triangle enclosing the drawing of G_μ without considering the translation of the poles of G_μ performed at ancestors of μ in T. The *reference point* of $\Delta(\mu)$ is the intersection of the right and bottom sides. The reference point of the drawing of G is conventionally positioned at (0, 0).

We consider a fully dynamic environment for the maintenance of Δ-drawings on a collection $\mathcal{G}$ of series-parallel digraphs. Each series-parallel digraph of $\mathcal{G}$ is represented by an SPQ-tree. We introduce the following set $\mathcal{O} = \mathcal{Q} \cup \mathcal{U}$ of operations:

- Query operations ($\mathcal{Q}$):
  - *Draw*(*vertex* v)—Return the (x, y) position of the drawing of vertex v of series-parallel digraph G.
  - *Offset*(*node* v)—Return the (x, y) position of the reference point of $\Delta(v)$, where v is a node in the SPQ-tree representing series-parallel digraph G.
  - *DrawSubgraph*(*node* v)—Draw the subgraph G_v as it appears in the drawing of G.

- *Window*(*node* v, *point* p, q)—Draw the portion of subgraph G_v contained in the query window defined by lower-left corner p and upper-right corner q.
- *Locate*(*node* v, *point* p)—Return the vertex, edge, or face of the digraph G_v containing point p.

- Update operations ($\mathcal{U}$):
  - *MakeDigraph*—Create a new elementary series-parallel digraph G, represented by a single Q-node, and add G to $\mathcal{G}$.
  - *DeleteDigraph*(*node* λ)—Remove from $\mathcal{G}$ the elementary series-parallel digraph represented by the single Q-node λ.
  - *Compose*(*type* X; *node* ρ', ρ'')—Perform a composition on the series-parallel digraphs $G_{\rho'}$ and $G_{\rho''}$. The composition is series or parallel according to whether $X = S$ or $X = P$. The resulting series-parallel digraph is added to $\mathcal{G}$ while $G_{\rho'}$ and $G_{\rho'}'$ are removed from $\mathcal{G}$.
  - *Attach*(*node* ρ, λ)—Replace the edge represented by Q-node λ with the series-parallel digraph G_ρ. The resulting series-parallel digraph is added to $\mathcal{G}$ while G_ρ is removed from $\mathcal{G}$.
  - *Detach*(*node* μ)—Remove the pertinent digraph G_μ of node μ from G and replace it with a single edge. The series-parallel digraph G_μ is added to $\mathcal{G}$.
  - *InsertEdge*(*vertex* v', v''; *edge* e)—Insert a new edge e from v' to v''. The operation is performed only if the resulting digraph is a series-parallel digraph.
  - *DeleteEdge*(*edge* e)—Delete edge e. The operation is performed only if the resulting digraph is a series-parallel digraph.
  - *InsertVertex*(*vertex* v; *edge* e, e', e'')—Replace edge e with two edges e' and e'' by inserting vertex v.
  - *DeleteVertex*(*vertex* v; *edge* e, e', e'')—Replace vertex v and its incident edges e' (incoming) and e'' (outgoing) with a single edge e. The operation is performed only if e' and e'' are the only incident edges of v.

In the rest of this section we show how to dynamically maintain a Δ-drawing and prove the following theorem.

THEOREM 3.2. *Consider the following dynamic graph drawing problem:*

- *Class of graphs $\mathcal{G}$: embedded series-parallel digraphs.*
- *Static drawing predicate $\mathcal{P}_S$: upward, planar, grid, straight-line, quasi-embedding-preserving, isomorphic, vertically symmetric, quadratic-area drawing.*
- *Repertory of operations $\mathcal{O}$: Draw, Offset, DrawSubgraph, Window, Locate, MakeDigraph, DeleteDigraph, Compose, Attach, Detach, InsertEdge, DeleteEdge, InsertVertex, and DeleteVertex.*
- *Dynamic drawing predicate $\mathcal{P}_D$: the drawing of a component not affected by an update operation changes only by a translation.*

There exists a fully dynamic algorithm for the above problem with the following performance:

- *A series-parallel digraph uses $O(n)$ memory space;*
- *Operations MakeDigraph and DeleteDigraph take each $O(1)$ time;*
- *Operation DrawSubgraph takes $O(\log n + k)$ time to return the position of k vertices and edges;*
- *Operation Window takes $O(k \cdot \log^2 n)$ time to return the position of k vertices and edges;*
- *Operations Draw, Offset, Compose, Attach, Detach, InsertEdge, DeleteEdge, InsertVertex, DeleteVertex, and Locate each take $O(\log n)$ time.*

In §3.3 we present a data structure for maintaining an implicit representation of a Δ-drawing. By Lemma 3.1, static predicate $\mathcal{P}_S$ is satisfied. In §3.4, we show how to perform

query operations. In §3.5, we show how to perform update operations and discuss the dynamic drawing predicate $\mathcal{P}_D$.

3.3. Data structure. If Z is a (x, y)-pair, then $x(Z)$ returns the x-value and $y(Z)$ returns the y-value. In order to maintain the Δ-drawing of a series-parallel digraph G represented by SPQ-tree T, we keep the following values for a node μ:

- *width*(μ)—The width of $\Delta(\mu)$. Note that the height of $\Delta(\mu)$ will be $2 \cdot$*width*(μ).
- *position*(μ)—The offset of the position of the reference point of $\Delta(\mu)$ from the position of the reference point of $\Delta(\Gamma)$.

The following values are relative positions in $\Delta(\mu)$ considering the reference point of $\Delta(\mu)$ to be at (0, 0):

- *sourceright*(μ)—The location of the drawing of the vertex connected to the rightmost outgoing edge from the source of G_μ.
- *sourceleft*(μ)—The location of the drawing of the vertex connected to the leftmost outgoing edge from the source of G_μ.
- *sinkright*(μ)—The location of the drawing of the vertex connected to the rightmost incoming edge to the sink of G_μ.
- *sinkleft*(μ)—The location of the drawing of the vertex connected to the leftmost incoming edge to the sink of G_μ.

The equations to calculate these values are linear expressions and are shown in Figs. 6–9. As an example, Fig. 10 shows pictorially how to calculate the value of *position* for the bounding triangle of a graph involved in a parallel composition. Note that if μ is the root of T, then *position*(μ) = (0, 0).

LEMMA 3.3. *Consider a node μ and an ancestor ν in SPQ-tree T. Then width(ν) can be expressed as width(μ) + d for some constant d. The value y(sourceright(ν)) can be expressed as*

$$a \cdot width(\mu) + b \cdot y(sourceright(\mu)) + c$$

for some a, b, c. The other values y(sourceleft(ν)), y(sinkright(ν)), and y(sinkleft(ν)) can be expressed similarly. Additionally, the value of y(position(μ)) can be expressed as

$$y(position(\nu)) + d'$$

for some constant d'. Similar equations hold for x-values.

Proof. By induction on the length of the path from node μ to node ν. If $\mu = \nu$ then the lemma is trivially true. The inductive step follows from the equations of Figs. 6, 7, 8, 9 and 10. □

For a solid or edge-path Π of T with head σ and tail τ, the transfer vector of Π is a vector of length 20 containing the constants in the equations of Lemma 3.3 associated with the calculations of the values *width*, *position*, *sourceright*, *sourceleft*, *sinkright*, and *sinkleft*.

We structure SPQ-tree T in the same manner as in §2, decomposing T into solid and edge paths. Suppose η is a path-tree node. If η is in the representation of a solid-path, we store at η the transfer vector of its associated subpath. We do not evert or reflect SPQ-trees, so we do not keep multiple versions of the transfer vector. If η is part of the representation of an edge-path, we store at η the value *width*(η) corresponding to the sum of the widths of the bounding triangles of the tree nodes associated with the nodes of $\Pi(\eta)$. We can maintain the transfer vectors under operation *join* in $O(1)$ time. This is immediate from the equations in Figs. 6–9.

$$
\begin{aligned}
width(\mu) &= \sum_{i=1}^{k} width(\mu_i), \\
x(position(\mu_j)) &= x(position(\mu)) + \sum_{i=j+1}^{k} width(\mu_i), \\
y(position(\mu_j)) &= y(position(\mu)) + \sum_{i=j+1}^{k} width(\mu_i) + \sum_{i=1}^{j-1} y(sourceright(\mu_i)), \\
x(sourceright(\mu)) &= 0, \\
y(sourceright(\mu)) &= \sum_{i=1}^{k} y(sourceright(\mu_i)), \\
x(sourceleft(\mu)) &= \sum_{i=2}^{k} width(\mu_i) + x(sourceleft(\mu_1)), \\
y(sourceleft(\mu)) &= \sum_{i=2}^{k} width(\mu_i) + y(sourceleft(\mu_1)), \\
x(sinkright(\mu)) &= 0, \\
y(sinkright(\mu)) &= \sum_{i=1}^{k-1} y(sourceright(\mu_i)) + y(sinkright(\mu_k)), \\
x(sinkleft(\mu)) &= \sum_{i=2}^{k} width(\mu_i) + x(sinkleft(\mu_1)), \\
y(sinkleft(\mu)) &= \sum_{i=2}^{k} width(\mu_i) + y(sinkleft(\mu_1)).
\end{aligned}
$$

FIG. 6. *The equations to calculate the values of width, sourceright, sourceleft, sinkright, and sinkleft for a P-node* μ *and the values of position for the children* μ_j, $1 \le j \le k$ *of* μ.

$$
\begin{aligned}
sourceright(\mu) &= (0, 2 \cdot width(\mu)), \\
sinkright(\mu) &= (0, 0).
\end{aligned}
$$

FIG. 7. *The equations to calculate the values of sourceright and sinkright for a* P_Q*-node* μ. *All other values are calculated as for a P-node (see Fig. 6).*

We keep the following additional values for each solid path Π from node σ to node τ.

- *samesource*(Π)—**true** if and only if the source of G_σ is the same vertex as the source of G_τ.
- *samesink*(Π)—**true** if and only if the sink of G_σ is the same vertex as the sink of G_τ.
- *noleftp*(Π)—**true** if and only if path Π does not contain both a P-node and its leftmost child.

Since the path-trees have a total of $O(n)$ nodes, and each node of the tree and of the path-trees uses $O(1)$ memory space, we conclude that the data structure uses $O(n)$ memory space.

$$
\begin{aligned}
width(\mu) &= \sum_{i=1}^{k} width(\mu_i), \\
x(position(\mu_j)) &= x(position(\mu)), \\
y(position(\mu_j)) &= y(position(\mu)) + 2 \cdot \sum_{i=1}^{j-1} width(\mu_i), \\
sourceright(\mu) &= sourceright(\mu_1), \\
sourceleft(\mu) &= sourceleft(\mu_1), \\
x(sinkright(\mu)) &= 0, \\
y(sinkright(\mu)) &= 2 \cdot \sum_{i=1}^{k-1} width(\mu_i) + y(sinkright(\mu_k)), \\
x(sinkleft(\mu)) &= x(sinkleft(\mu_k)), \\
y(sinkleft(\mu)) &= 2 \cdot \sum_{i=1}^{k-1} width(\mu_i) + y(sinkleft(\mu_k)).
\end{aligned}
$$

FIG. 8. *The equations to calculate the value of width, sourceright, sourceleft, sinkright, and sinkleft for an S-node μ and the value of position for children μ_j, $1 \le j \le k$ of μ.*

$$
\begin{aligned}
width(\mu) &= 1, \\
sourceright(\mu) &= (0, 2), \\
sourceleft(\mu) &= (0, 2), \\
sinkright(\mu) &= (0, 0), \\
sinkleft(\mu) &= (0, 0).
\end{aligned}
$$

FIG. 9. *The values of width, sourceright, sourceleft, sinkright, and sinkleft for a Q-node μ.*

We use the following auxiliary operation to find proper nodes:

• *Proper*(*vertex* v)—Returns the node triple (μ, μ', μ''), where μ is the proper node of v, node μ' is the child of v such that v is the sink of G'_μ, and μ'' is the child of v such that v is the source of G''_μ. If v is a pole of G, then *Proper*(v) returns $(\rho, -, -)$, where ρ is the root of T.

We use the following path selection function to find *Proper*(v). Selection function S_4 takes as an argument integer x which has value either 1 or 2 indicating if v is the source or sink of the pertinent digraph of the head of $\Pi(\eta)$.

Selection function S_4. *Suppose η is the root of a path-tree with children η' and η''. Let Π' be the concatenation of the subpath represented by η' with the node head(η''). If $x = 1$ and samesource(Π') =* **true** *or if $x = 2$ and samesink(Π') =* **true** *then return η''. Otherwise return η'.*

When used with operation *PathFind*, path selection function S_4 returns the tail of the longest subpath beginning at *head*(η) such that the pole of *head*(η) indicated by x is also a pole of the returned node. Therefore, the parent of the returned node will be the proper node of v.

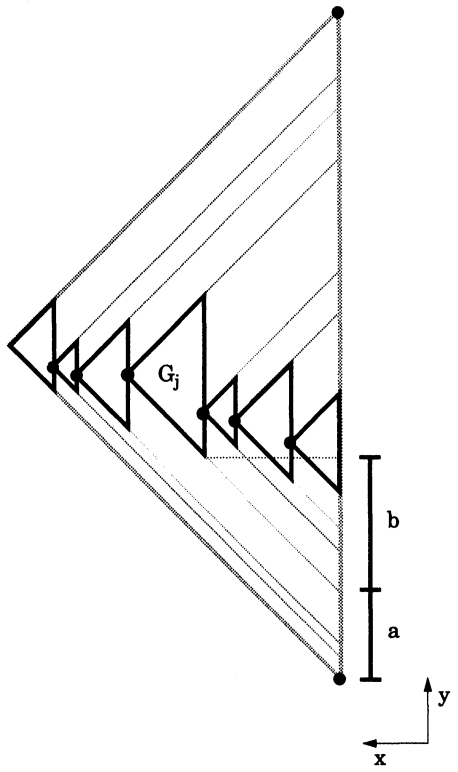


FIG. 10. *A pictorial representation of the calculations of position(μ_j), where μ_j is a child of a P-node μ. Graph G_j is the pertinent graph of μ_j. Notice lengths $a = \sum_{i=1}^{j-1} y(sourceright(\mu_i))$ and $b = \sum_{i=j+1}^{k} width(\mu_i)$.*

If v is the source or sink of G, then return $(\rho, -, -)$, where ρ is the root of T. Otherwise, let e be any edge such that v is the source of e and let node λ be the associated Q-node. We find node μ'' by calling *PathFind*(λ, S_4). Nodes μ and μ' are the parent and left sibling of μ''. This takes $O(\log n)$ time.

3.4. Query operations. Operation *Offset*(v) is implemented as for trees. We call *expose*(v), return the value of *position*(v), then restore the size invariant by calling *conceal*(v).

Operation *Draw*(v) is performed as follows. Find *Proper*(v). If v is the source of G, then return (0, 0). If v is the sink of G then return $(0, 2 \cdot width(\rho))$, where ρ is the root of T. Otherwise, suppose *Proper*(v) = (μ, μ', μ''). Return *Offset*(μ''). This can all be done in $O(\log n)$ time.

Operation *DrawSubgraph*(v) is performed by calling *Draw* on the poles of G_v and then calling *Offset*(v) in order to determine the position of $\Delta(v)$. We then visit G_v and compute the positions in $O(1)$ amortized time per vertex and edge using the sequential Δ-algorithm (see Lemma 3.1).

Each internal face in a drawing Γ of series-parallel digraph G can be associated with the parallel composition of two subgraphs. Therefore, we identify each internal face f of Γ by the node triplet (v, μ_ℓ, μ_r), where v is a P-node or a P_Q-node and μ_ℓ and μ_r are the consecutive children of v such that f is associated with the parallel composition of G_{μ_ℓ} and G_{μ_r}, and G_{μ_ℓ} embedded to the left of G_{μ_r}. Operation *Locate*(v, p) consists of the following steps for a digraph G represented by SPQ-tree T:

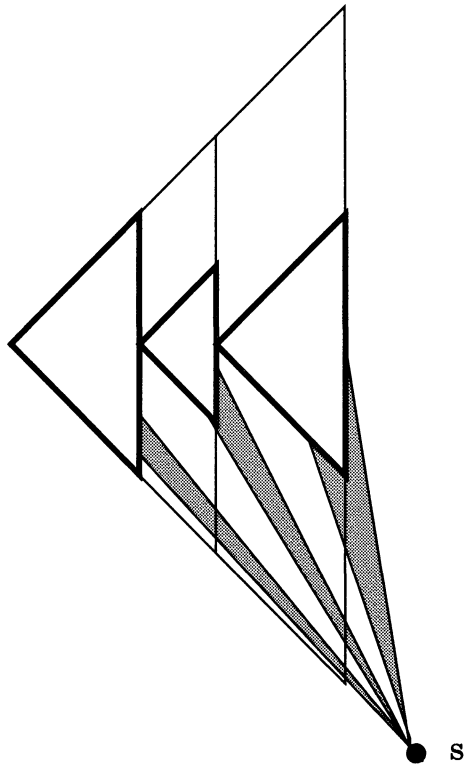


FIG. 11. *Finding the face containing a query point p. If p is in a shaded region then its face is found at a descendant P-node.*

(1) If point p is not contained in $\Delta(v)$, then return the external face.

(2) Find the deepest descendant μ of v such that p is contained in $\Delta(\mu)$, but not the bounding triangle of any of the children of μ.

(3) Let s and t be the source and sink of G_μ. If p is at *Draw*(s) or *Draw*(t) then return the found vertex. If p is on the rightmost edge from s or the rightmost edge entering t, then return the edge.

(4) If node μ is an S- or Q-node, then p is to the left of the drawing of any of the edges of G_μ. Let node v' be the closest ancestor of μ such that node μ is not a descendant of the leftmost child of v', let node μ'' be the child of v' on the path from μ to v', and let node μ' be the left sibling of μ''. Therefore, the face containing p will be (v', μ', μ'').

(5) The final case is if node μ is a P-node or a P_Q-node. Let $R_s(\mu)$ be the region bounded by the triangle formed by the drawing of vertices s, *sourceleft*(μ), and *sourceright*(μ). Similarly, let $R_t(\mu)$ be the region formed by the drawing of vertices t, *sinkleft*(μ), and *sinkright*(μ).

(a) If p is not contained in $R_s(\mu)$ or $R_t(\mu)$, then, as above, p is to the left of the drawing of any of the edges of G_μ, and we continue as in step 2.

(b) Otherwise, suppose p is in $R_s(\mu)$ (the case where p is in $R_t(\mu)$ is handled similarly). Let node v', be the deepest descendant of μ such that point p is contained in $R_s(v')$. Node v' will be a P-node, since for S-node κ with left-child κ', $R_s(\kappa) = R_s(\kappa')$. If p is on an edge of $R_s(\mu)$, then return the edge. Otherwise, the face containing p is (v', μ', μ''), where μ' is the child of v' such that point p is to the right of $R_s(\mu')$ but to the left of $R_s(\mu'')$ where node μ'' is the immediate left sibling of μ' (see Fig. 11).

Steps 2, 3, and 5b are implemented using selection functions. Consider the following tree selection functions, which take query point p as an argument.

Selection function S_5. *Suppose η is the root of a path-tree with children η' and η''. If point p is contained in $\Delta(tail(\eta'))$ then return η', otherwise return η''.*

We then perform step 2 by letting node $\kappa = TreeFind(\nu, S_5, p)$. If κ is a node on an edge-path of node κ', then return κ'. Otherwise, return node κ.

Selection function S_6. *Suppose η is the root of a path-tree with children η' and η''. If point p is contained in $R_s(tail(\eta'))$ then return η', else return η''.*

We then perform step 5b by letting node $\kappa = TreeFind(\mu, S_6, p)$. If κ is node $\nu'(\mu'')$ on an edge-path of node ν', then point p is on the face (ν', μ', μ''), where node μ' is the left-sibling of μ''. Otherwise, node κ is on a solid path Π. Let node μ' be the child of κ on Π. If p is on an edge of $R_s(\kappa)$, then return the edge. If p is to the right of the edge from the source of G_κ to $sourceright(\mu')$. Then p is contained in face (κ, μ', μ''), where μ'' is the right-sibling of μ'. Otherwise, point p is to the left of the edge from the source of G_κ to $sourceleft(\mu')$. Then p is contained in face (κ, μ''', μ'), where μ''' is the left-sibling of μ'.

Consider the following path selection function.

Selection function S_7. *Suppose η is the root of a path-tree with children η' and η''. If $noleftp(\eta')$ is* **true** *or if $head(\eta'')$ is a P-node and $tail(\eta')$ is not its leftmost child, then return η', else return η''.*

Step 4 is implemented by first calling operation *expose* to get path Π from μ to ν. Let node $\kappa'' = PathFind(\Pi, S_7)$. If $\kappa'' = \nu$ then return the external face. Otherwise, return the node-triple $(\kappa, \kappa', \kappa'')$, where nodes κ and κ' are the parent and left-sibling of node κ''.

Each of these steps can be implemented in $O(\log n)$ time. Therefore, operation *Locate* is performed in $O(\log n)$ time.

To implement operation *Window* we keep the data structure of [41] to maintain the planar embedding of G. In particular given a face f of G in an upward embedding of series-parallel digraph G, we can find two lists of edges and vertices that comprise the left and right boundary of f.

Suppose p and q are points with $x(p) < x(q)$ and $y(p) < y(q)$, and let W be the window defined by p and q. Operation $Window(\nu, p, q)$ is realized as follows. If W does not intersect $\Delta(\nu)$ then return an empty drawing. Otherwise, let s be a scan point. Initialize s to p, and do the following, clipping to W:

- Perform $Locate(\nu, s)$. Let f be the returned face. Find the edge e that is to the right of s on the boundary of f. Draw and mark all unmarked edges and vertices that are in W and are reachable by forward and reverse edges from edge e.
- Repeat step 1, continuing around the boundary of W, finding unmarked edges of G that intersect the boundary of W.

We implement the search of the list for face f in step 1 by performing a binary search on the location of the vertices on the boundary. Each *Draw* call takes $O(\log n)$ time, so finding an edge that intersects the boundary of W takes $O(\log^2 n)$ time. We traverse the edges reachable from e by using a modified breadth-first search that cuts at a marked vertex, or at the boundary of W. Hence, the internal j edges and vertices found by step 1 can be drawn in $O(j \cdot \log n)$ time.

Therefore, operation *Window* can be implemented in $O(k \cdot \log^2 n)$ time to draw k vertices and edges.

3.5. Update operations. Operations *MakeDigraph* and *DeleteDigraph* can be trivially implemented in $O(1)$ time. Operations *Compose*, *Attach*, and *Detach* can each be implemented with a constant number of calls to *MakeDigraph*, *DeleteDigraph*, and variations of the tree operations *Link* and *Cut*.

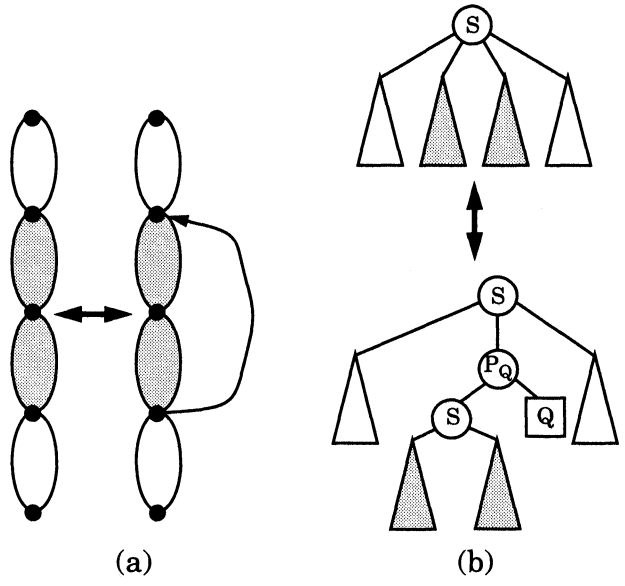


FIG. 12. *Modification of T in operation InsertEdge and DeleteEdge: splitting/merging an S-chain.* (a) *Insertion/deletion of an edge between nonconsecutive vertices of the skeleton of an S-chain.* (b) *Transformation of the SPQ-tree.*

Consider now operation *InsertVertex*(v, e, e', e'') and let λ be the Q-node of e, and v be the parent of λ, if it exists. We replace λ with an S-node μ having Q-node children for e' and e''. This can be done with two *MakeDiagraph* operations, followed by a *Compose* operation and one each *Cut* and *Link* operation. Then, if μ is an S-node, we contract μ into v. All this takes $O(\log n)$ time.

The inverse operation *DeleteVertex*(e, v, e', e'') is implemented similarly with a constant number of *MakeDigraph*, *DeleteDigraph*, *Link*, *Cut*, and *Expand* operations.

The restructuring of the SPQ-tree caused by *InsertEdge* and *DeleteEdge* is more complex. It is known [6] that if G is a series-parallel digraph with SPQ-tree T and v' and v'' are two vertices of G, then the digraph obtained by inserting an edge from v' to v'' is a series-parallel digraph if and only if either $v' = s$ and $v'' = t$, or v' and v'' are vertices of the skeleton of the same S-node of T, with v' preceding v''. If e is an edge of G represented by Q-node λ, then the digraph obtained by removing edge e is a series-parallel digraph if and only if one of the following conditions is true: the parent of λ in T is a P_Q-node, e is the only outgoing edge of s, or e is the only incoming edge of t. These conditions can each be tested in $O(\log n)$ time. The restructuring to T required for *InsertEdge* and *DeleteEdge* also can be performed in $O(\log n)$ time.

The transformations to T that result from *InsertEdge* and *DeleteEdge* are demonstrated in Figs. 12 and 13. Notice that transitive edges are always added as the right child of a P_Q-node.

In a Δ-drawing, the drawing of a component is independent, up to a translation, from the structure of the rest of the digraph [2]. Hence, dynamic predicate $\mathcal{P}_D$ is satisfied.

4. Planar graphs. In this section we present dynamic techniques for drawing planar graphs. First we discuss upward drawings of planar *st*-digraphs, and next we extend the results to (undirected) biconnected planar graphs. Planar *st*-digraphs, which include series-parallel digraphs as a special case, were first introduced by Lempel, Even, and Cederbaum [26] in connection with a planarity testing algorithm, and they have subsequently been used

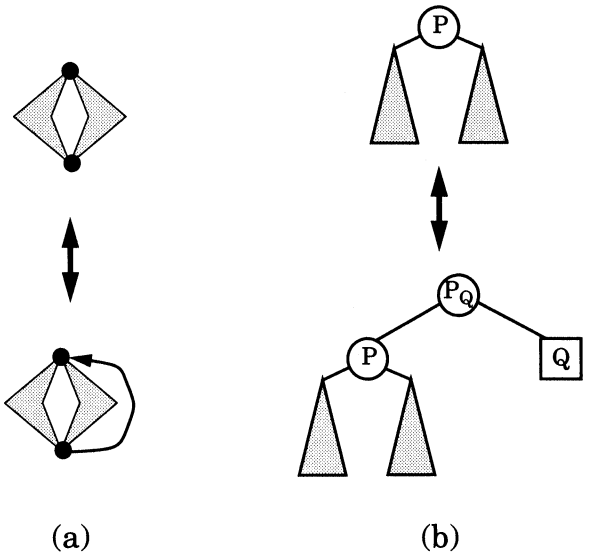


FIG. 13. *Modification of T in operation InsertEdge and DeleteEdge: extending/shortening a P-chain.* (a) *Insertion/deletion of an edge between consecutive vertices of the skeleton of an S-chain (the vertices are the source and sink of a parallel composition).* (b) *Transformation of the SPQ-tree.*

in several applications, including planar graph embedding [5], [10], [41], graph drawing [9], [11], and planar point location [16], [20], [31], [47].

A *planar st-digraph* is a planar acyclic directed graph with exactly one source vertex s and exactly one sink vertex t, which is embedded in the plane such that s and t are on the boundary of the external face.

The following generalizes our definition of components of series-parallel digraphs. A digraph is *weakly connected* if its underlying undirected graph is connected. Let G by a planar st-digraph. An *open component* of G is a maximal weakly connected subgraph G' of the digraph obtained from G by removing a separation pair $\{p, q\}$, such that G' does not contains s or t. A *closed component* of G is an induced subgraph G' of G such that

- G' is a planar pq-digraph;
- G' contains every vertex of G that is on some path from p to q;
- G' contains every outgoing edge of p, every incoming edge of q, and every incident edge of the remaining vertices of G'.

A *component* of G is either a closed or an open component.

4.1. Upward drawings. We consider the following static drawing predicate $\mathcal{P}_S$:

Upward: The drawing is upward.

Planar: The drawing is planar.

Embedding-preserving: The drawing preserves the embedding.

Grid: Vertices are placed at integer coordinates.

Polyline: Edges are drawn as polygonal lines.

Transitive-bends: Transitive edges have at most one bend. The other edges are straight lines. Hence the total number of bends is at most $2n - 5$.

Isomorphic: Isomorphic components have drawings that are congruent up to a translation.

Symmetric: Symmetric components have drawings that are congruent up to a translation and a reflection.

Quadratic-area: The drawing has $O(n^2)$ area.

We dynamize the polyline drawing method of [11], which has the important property of displaying symmetries and isomorphisms of subgraphs. Note that we do not consider straight-line drawings because they may require exponential area [11].

4.1.1. Dymamic environment. We consider a fully dynamic environment for the maintenance of upward drawings on a collection of embedded a planar *st*-digraphs. Namely, we introduce the following set $\mathcal{O} = \mathcal{Q} \cup \mathcal{U}$ of operations:

- Query operations ($\mathcal{Q}$):
  - *Draw*(*vertex* v)—Return the (x, y) position of vertex v. The source is considered to be at (0, 0).
  - *Draw*(*edge* e)—Return the (x, y) position of the endpoints of edge e. If e is a transitive edge, then also return the position of the bend of e.
- Update operations ($\mathcal{U}$):
  - *MakeDigraph*—Create a new elementary planar *st*-digraph G, consisting of a single edge.
  - *DeleteDigraph*(*edge* e)—Remove the elementary planar *st*-digraph consisting of single edge e.
  - *InsertEdge*(*vertex* v', v''; *edge* e; *face* f, f', f'')—Add edge $e = (v', v'')$ inside face f, which is decomposed into faces f' and f'', with f' to the left of e and f'' to the right.
  - *DeleteEdge*(*edge* e; *face* f)—Delete edge e and merge the two faces formerly on the two sides of e into a new face f.
  - *Expand*(*vertex* v, v', v''; *edge* e; *face* f', f'')—Expand vertex v into vertices v' and v'', which are connected by a new edge e with face f' to its left and face f'' to its right.
  - *Contract*(*vertex* v; *edge* e)—Contract edge e and merge its endpoints into a new vertex v. Parallel edges resulting from the contraction are merged into a single edge.

Each update operation is allowed if the resulting digraph is itself a planar *st*-digraph.

Consider a vertex v of planar *st*-digraph G. Let $R^+(v)$ be the set of vertices of G reachable from v and let $R^-(v)$ be the set of vertices u of G such that v is reachable from u. For a pair of vertices (v', v'') the *subgraph of stable reachability* of (v', v'') is the subgraph induced by $R^-(v') \cup R^+(v'')$.

In the rest of this section we prove the following theorem.

THEOREM 4.1. *Consider the following dynamic graph drawing problem:*

- *Class of graphs $\mathcal{G}$: embedded planar st-digraphs.*
- *Static drawing predicate $\mathcal{P}_S$: upward, planar, embedding-preserving, grid, polyline, transitive-bends, isomorphic, symmetric, quadratic-area drawing.*
- *Repertory of operations $\mathcal{O}$: Draw, MakeDigraph, DeleteDigraph, InsertEdge, DeleteEdge, Expand, and Contract.*
- *Dynamic drawing predicate $\mathcal{P}_D$:*
  - *The drawing of a component not affected by an update operation changes only be a translation;*
  - *After inserting or deleting an edge between v' and v'', the drawing of the subgraph of stable reachability of (v', v'') changes only by a translation.*

There exists a fully dynamic algorithm for the above problem with the following performance:

- *A planar st-digraph uses $O(n)$ memory space;*
- *Operations MakeDigraph and DeleteDigraph each take $O(1)$ time;*
- *Operations Draw, InsertEdge, DeleteEdge, Expand, and Contract each take $O(\log n)$ time.*

Data structure. Let V be the set of vertices E, be the set of edges, and F be the set of faces of planar st-digraph G. As shown in [11], [42], there are two orderings on the set $V \cup E \cup F$, denoted L and R, such that if G has no transitive edges, a planar upward grid drawing of G is obtained by assigning to each vertex v x- and y-coordinates equal to the ranks of v in the restriction to V of L and R, respectively. This drawing method is extended to general planar st-digraphs by inserting a dummy vertex (a bend) along each transitive edge.

We represent sequences L and R by means of balanced binary trees T_L and T_R (e.g., using red-black trees [21]). Each leaf represents a vertex, edge, or face. At each leaf, we keep the following binary value: 1 if the node is associated with a vertex or a transitive edge and 0 otherwise. At each internal node η, we store the sum of the values at the leaves in the subtree of η. In a drawing query, we compute $x(v)$ (resp., $y(v)$) by splitting tree T_L (resp., T_R) at the node associated with v, and finding the value stored at the root of the resulting left tree.

After an update operation at most two edges of G become or cease to be transitive, and each such edge can be identified in $O(\log n)$ time. The corresponding modifications of node values can be done in $O(\log n)$ time. Also, sequences L and R are updated by means of $O(1)$ split/concatenate operations [42], so that the corresponding updates on T_L and T_R take $O(\log n)$ time. We conclude that our dynamic data structure uses $O(n)$ memory space and supports each operation in $O(\log n)$ time.

The dynamic drawing predicate $\mathcal{P}_D$ is satisfied because of the results in [11].

4.2. Visibility drawings. The concept of *visibility* plays a fundamental role in a variety of geometric problems and applications, such as art gallery problems [30], very large-scale integration layout [23], [34], [50], motion planning [22], and graph drawing [9], [44]. A *visibility representation* Θ for a directed graph G maps each vertex v of G to a horizontal segment $\Theta(v)$ and each edge (u, v) to a vertical segment $\Theta(u, v)$ that has its lower endpoint on $\Theta(u)$, its upper endpoint on $\Theta(v)$, and does not intersect any other horizontal segment. Besides having many applications, visibility representations are also of intrinsic theoretical interest, and their combinatorial properties have been extensively investigated [13], [43], [45], [51], [52].

We consider the following static drawing predicate $\mathcal{P}_S$:

Visibility: The drawing is a visibility representation.

Grid: The endpoints of vertex- and edge-segments are placed at integer coordinates.

Isomorphic: Isomorphic components have drawings that are congruent up to a translation.

Quadratic-area: The drawing has $O(n^2)$ area.

In the rest of this section we prove the following theorem:

THEOREM 4.2. *Consider the following dynamic graph drawing problem:*

- *Class of graphs $\mathcal{G}$: embedded planar st-digraphs.*
- *Static drawing predicate $\mathcal{P}_S$: visibility, grid, isomorphic, quadratic-area drawing.*
- *Repertory of operations $\mathcal{O}$: Draw, MakeDigraph, DeleteDigraph, InsertEdge, DeleteEdge, Expand, and Contract.*
- *Dynamic drawing predicate $\mathcal{P}_D$: the drawing of an open component not affected by an update operation changes only by a translation.*

There exists a fully dynamic algorithm for the above problem with the following performance:

- *A planar st-digraph uses $O(n)$ memory space;*
- *Operations MakeDigraph and DeleteDigraph each take $O(1)$ time;*
- *Operations Draw, InsertEdge, DeleteEdge, Expand, and Contract each take $O(\log n)$ time.*

Data structure. We recall that in a planar st-digraph the incoming edges of each vertex appear consecutively around the vertex, and so do the outgoing edges [43]. The faces separating the incoming and outgoing edges of vertex v to the left and right of v are called *left*(v) and *right*(v), respectively. Also, the boundary of each face f consists of two directed paths enclosing f, each starting from the unique lowest vertex *low*(f) and ending at the unique highest vertex *high*(f). A visibility representation for G can be constructed by the following variation of previous sequential algorithms [9], [33], [43].

(1) Construct the directed dual of planar st-digraph G as follows. (a) Construct the dual graph G^* of G. (b) Orient the dual of each edge e of G from the face to the left of e to the face to the right of e. (c) Expand the vertex of G^* associated with the external face of G into two vertices, denoted s^* and t^*, between faces s and t of G^*. (d) Remove edge (t^*, s^*) of G^* and let D be the resulting planar s^*t^*-digraph.

(2) Compute a topological ordering $Y(v)$ of the vertices of G.

(3) Compute a topological ordering $X(f)$ of the vertices of D.

(4) Draw each vertex-segment $\Theta(v)$ at y-coordinate $Y(v)$ and between x-coordinates $X(left(v))$ and $X(right(v)) - 1$.

(5) Draw each edge-segment $\Theta(e)$ at x-coordinate $X(left(e))$ and between y-coordinates $Y(low(e))$ and $Y(high(e))$.

Consider the orderings L and R defined in §4.2. The restriction of sequence L (or R) to V is a topological ordering [42]. Hence, we can use balanced binary trees to dynamically maintain topological ordering X and Y, such that the position of a vertex- or edge-segment can be computed in $O(\log n)$ time.

The dynamic drawing predicate $\mathcal{P}_D$ is satisfied because of the results in [11].

4.3. Biconnected planar graphs. Finally, we extend our results to (undirected) biconnected planar graphs. We consider the following static drawing predicate $\mathcal{P}_S$:

Planar: The drawing is planar.

Embedding-preserving: The drawing preserves the embedding.

Grid: Vertices are placed at integer coordinates.

Polyline: Edges are drawn as polygonal lines.

One-bend: Each edge has at most one bend and the total number of bends is at most $2n - 5$.

Quadratic-area: The drawing has $O(n^2)$ area.

We consider a semidynamic environment for the maintenance of polyline drawings on a collection of biconnected planar graphs. Namely, we introduce the following set $\mathcal{O} = \mathcal{Q} \cup \mathcal{U}$ of operations:

- Query operations ($\mathcal{Q}$):
  - *Draw*(*vertex* v)—Return the (x, y) position of vertex v.
  - *Draw*(*edge* e)—Return the (x, y) position of the endpoints of edge e. If e has a bend, then also return the position of the bend.
- Update operations ($\mathcal{U}$):
  - *MakeGraph*—Create a new elementary biconnected planar graph G, consisting of a cycle with three vertices.
  - *InsertEdge*(*vertex* v', v''; *edge* e; *face* f, f', f'')—Add edge $e = (v', v'')$ inside face f, which is decomposed into faces f' and f''.
  - *InsertVertex*(*vertex* v; *edge* e, e', e'')—Insert vertex v on edge e, which is decomposed into edges e' and e''.

As shown in [10], this repertory of operation is complete; i.e., any n-vertex biconnected planar graph can be assembled by means of $O(n)$ operations of the repertory. In the rest of this section we prove the following theorem:

THEOREM 4.3. *Consider the following dynamic graph drawing problem:*

- *Class of graphs $\mathcal{G}$: biconnected planar graphs.*
- *Static drawing predicate $\mathcal{P}_S$: planar, embedding-preserving, grid, polyline, one-bend, quadratic-area drawing.*
- *Repertory of operations $\mathcal{O}$: Draw, MakeGraph, InsertEdge, and InsertVertex. There exists a semidynamic algorithm for the above problem with the following performance:*
- *A biconnected planar graph uses $O(n)$ memory space;*
- *Operation MakeDigraph takes $O(1)$ time;*
- *Operations Draw, InsertEdge, and InsertVertex each take $O(\log n)$ time.*

Note that we do not maintain a dynamic drawing predicate.

Data structure. We maintain an on-line orientation of G into a planar st-digraph. This can be done using the techniques of [41].

We can extend Theorem 4.3 to support the insertion of an edge between two vertices that are not on the same face of the current embedding, using the techniques of [10]. In this case the embedding has to be modified in order to preserve planarity, and the time complexity of operation *InsertEdge* is amortized.

With a similar approach, we can derive from the data structure of Theorem 4.2 a semidynamic data structure for maintaining on-line visibility representations of biconnected planar graphs. The memory space and time complexity is the same as in Theorem 4.3.

5. Open problems. Open problems include the following:

- Decrease the complexity of window queries in trees and series-parallel digraphs to $O(k + \log n)$.
- Extend the techniques for planar st-digraphs and general planar graphs to support point-location and window queries.
- Develop dynamic algorithms for planar straight-line drawings of general planar graphs. The techniques of [18], [35] appear difficult to dynamize.
- Dynamically maintain planar orthogonal drawings with the minimum number of bends. The static algorithm of [40] is based on network flow techniques for which no dynamic methods are known.
- Devise dynamic algorithms to test whether a digraph admits an upward planar drawing. Static algorithms that perform this test are known only for triconnected [3], bipartite [8], and single-source digraphs [4], [24]. Semidynamic planarity testing can be done with $O(\log n)$ query and insertion time [10]. Recently, a fully dynamic planarity testing technique with $O(n^{2/3})$ query and update time has been discovered [19].
- Dynamize drawing methods for general graphs that are based on physical models of the layout process, such as the "spring" algorithm [14], [25].

Acknowledgment. We are grateful to Paola Bertolazzi for helpful discussions and insights.

REFERENCES

[1] S. W. BENT, D. D. SLEATOR, AND R. E. TARJAN, *Biased search trees*, SIAM J. Comput., 14 (1985), pp. 545–568.

[2] P. BERTOLAZZI, R. F. COHEN, G. DI BATTISTA, R. TAMASSIA, AND I. G. TOLLIS, *How to draw a series-parallel digraph*, Internat. J. Comput. Geom. Appl, to appear.

[3] P. BERTOLAZZI AND G. DI BATISTA, *On upward drawing testing of triconnected digraphs*, Algorithmica, to appear.

[4] P. BERTOLAZZI, G. DI BATTISTA, C. MANNINO, AND R. TAMASSIA, *Optimal upward planarity testing of single-source digraphs*, in Proc. European Symp. on Algorithms, Lecture Notes in Computer Science 726, Springer-Verlag, 1993, pp. 37–48.

[5] N. CHIBA, T. NISHIZEKI, S. ABE, AND T. OZAWA, *A linear algorithm for embedding planar graphs using PQ-Trees*, J. Comput. System Sci., 30 (1985), pp. 54–76.

[6] R. F. COHEN AND R. TAMASSIA, *Dynamic expression trees and their applications*, in Proc. ACM-SIAM Symp. on Discrete Algorithms, 1991, pp. 52–61.

[7] G. DI BATTISTA, P. EADES, R. TAMASSIA, AND I. G. TOLLIS, *Algorithms for Automatic Graph Drawing: An Annotated Bibliography*, Technical Report, Dept. of Computer Science, Brown Univ., Providence, RI. Available via anonymous ftp from wilma.cs.brown.edu. (128.148.33.66), files /pub/ftp/papers/compgeo/gdbiblio.tex.Z and /pub/ftp/papers/compgeo/gdbiblio.ps.Z., 1993.

[8] G. DI BATTISTA, W.-P. LIU, AND I. RIVAL, *Bipartite graphs, upward drawings, and planarity*, Informa. Proces. Lett., 36 (1990), pp. 317–322.

[9] G. DI BATTISTA AND R. TAMASSIA, *Algorithms for plane representations of acyclic digraphs*, Theoret. Comput. Sci., 61 (1988), pp. 175–198.

[10] G. DI BATTISTA AND R. TAMASSIA, *Incremental planarity testing*, in Proc. 30th IEEE Symp. on Foundations of Computer Science, 1989, pp. 436–441.

[11] G. DI BATTISTA, R. TAMASSIA, AND I. G. TOLLIS, *Area requirement and symmetry display of planar upward drawings*, Discrete Comput. Geom. 7 (1992), pp. 381–401.

[12] D. DOLEV, F. T. LEIGHTON, AND H. TRICKEY, *Planar embedding of planar graphs*, in Advances in Computing Research, Vol. 2, F. P. Preparata, ed., JAI Press Inc., Greenwich, CT, 1984, pp. 147–161.

[13] P. DUCHET, Y. HAMIDOUNE, M. LAS VERGNAS, AND H. MEYNIEL, *Representing a planar graph by vertical lines joining different levels*, Discrete Math. 46 (1983), pp. 319–321.

[14] P. EADES, *A heuristic for graph drawing*, Congr. Numer., 42 (1984), pp. 149–160.

[15] P. EADES AND K. SUGIYAMA, *How to draw a directed graph*, J. Inform. Process., 13 (1991), pp. 424–437.

[16] H. EDELSBRUNNER, L. J. GUIBAS, AND J. STOLFI, *Optimal point location in a monotone subdivision*, SIAM J. Comput., 15 (1986), pp. 317–340.

[17] M. FORMANN, T. HAGERUP, J. HARALAMBIDES, M. KAUFMANN, F. T. LEIGHTON, A. SIMVONIS, E. WELZL, AND G. WOEGINGER, *Drawing graphs in the plane with high resolution*, in Proc. IEEE Symp. on Foundations of Computer Science, 1990, pp. 86–95.

[18] H. DE FRAYSSEIX, J. PACH, AND R. POLLACK, *How to draw a planar graph on a grid*, Combinatorica, 10 (1990), pp. 41–51.

[19] Z. GALIL, G. F. ITALIANO, AND N. SARNAK, *Fully dynamic planarity testing*, in Proc. 24th ACM Symp. on Theory of Computing, 1992, pp. 495–506.

[20] M. T. GOODRICH AND R. TAMASSIA, *Dynamic trees and dynamic point location*, in Proc. 23th ACM Symp, on Theory of Computing, 1991, pp. 523–533.

[21] L. J. GUIBAS AND R. SEDGEWICK, *A dichromatic framework for balanced trees*, in Proc. 19th IEEE Symp. on Foundations of Computer Science, 1978, pp. 8–21.

[22] L. J. GUIBAS AND F. F. YAO, *On translating a set of rectangles*, in Advances in Computing Research, Vol. 1, F. P. Preparata, ed., JAI Press Inc., Greenwich, CT, 1983, pp. 61–77.

[23] M. Y. HSUEH AND D. O. PEDERSON, *Computer-aided layout of LSI circuit building-blocks*, in Proc. IEEE Int. Symp. on Circuits and Systems, 1979, pp. 474–477.

[24] M. D. HUTTON AND A. LUBIW, *Upward planar drawing of single source acyclic digraphs*, Proc. ACM-SIAM Symp. on Discrete Algorithms, 1991, pp. 203–211.

[25] T. KAMADA, *Visualizing Abstract Objects and Relations*, World Scientific, Teaneck, NJ, 1989.

[26] A. LEMPEL, S. EVEN, AND I. CEDERBAUM, *An algorithm for planarity testing of graphs*, in Theory of Graphs, Int. Symposium (Rome, 1966), Gordon and Breach, New York, 1967, pp. 215–232.

[27] S. M. MALITZ AND A. PAPAKOSTAS, *On the angular resolution of planar graphs*, in Proc. ACM Symp. on Theory of Computing, 1992, pp. 527–538.

[28] S. MOEN, *Drawing dynamic trees*, IEEE Software, 7 (1990), pp. 21–28.

[29] T. NISHIZEKI AND N. CHIBA, *Planar Graphs: Theory and Algorithms*, Annals of Discrete Mathematics 32, North-Holland, Amsterdam, 1988.

[30] J. O'ROURKE, *Art Gallery Theorems and Algorithms*, Oxford University Press, London, 1987.

[31] F. P. PREPARATA AND R. TAMASSIA, *Fully dynamic point location in a monotone subdivision*, SIAM J. Comput., 18 (1989), pp. 811–830.

[32] E. REINGOLD AND J. TILFORD, *Tidier drawing of trees*, in IEEE Trans. on Software Engineering SE-7, 1981, pp. 223–228.

[33] P. ROSENSTIEHL AND R. E. TARJAN, *Rectilinear planar layouts of planar graphs and bipolar orientations*, Discrete Comput. Geom., 1 (1986), pp. 343–353.

[34] M. SCHLAG, F. LUCCIO, P. MAESTRINI, D. T. LEE, AND C. K. WONG, *A visibility problem in VLSI layout compaction*, in Advances in Computing Research, Vol. 2, F. P. Preparata, ed., JAI Press Inc., Greenwich, CT, 1985, pp. 259–282.

[35] W. SCHNYDER, *Embedding planar graphs on the grid*, in Proc. ACM-SIAM Symp. on Discrete Algorithms, 1990, pp. 138–148.

[36] D. D. SLEATOR AND R. E. TARJAN, *A data structure for dynamic trees*, J. Comput. System Sci., 24 (1983), pp. 362–381.

[37] D. D. SLEATOR AND R. E. TARJAN, *Self-adjusting binary search trees*, J. Assoc. Comput. Mach., 32 (1985), pp. 652–686.

[38] K. SUGIYAMA, S. TAGAWA, AND M. TODA, *Methods for visual understanding of hierarchical systems*, in IEEE Trans. on Systems, Man, and Cybernetic SMC-11, 1981, pp. 109–125.

[39] K. J. SUPOWIT AND E. M. REINGOLD, *The complexity of drawing trees nicely*, Acta Inform., 18 (1983), pp. 377–392.

[40] R. TAMASSIA, *On embedding a graph in the grid with the minimum number of bends*, SIAM J. Comput., 16 (1987), pp. 421–444.

[41] R. TAMASSIA, *A dynamic data structure for planar graph embedding*, in Automata, Languages and Programming (Proc. 15th ICALP), Lecture Notes in Computer Science 317, Springer, Berlin, New York, 1988, pp. 576–590.

[42] R. TAMASSIA AND F. P. PREPARATA, *Dynamic maintenance of planar digraphs, with applications*, Algorithmica, 5 (1990), 509–527.

[43] R. TAMASSIA AND I. G. TOLLIS, *A unified approach to visibility representations of planar graphs*, Discrete Comput. Geom., 1 (1986), pp. 321–341.

[44] R. TAMASSIA AND I. G. TOLLIS, *Planar grid embedding in linear time*, in IEEE Trans. on Circuits and Systems CAS-36, 1989, pp. 1230–1234.

[45] R. TAMASSIA AND I. G. TOLLIS, *Tessellation representation of planar graphs*, in Proc. 27th Annual Allerton Conf., 1989, pp. 48–57.

[46] R. TAMASSIA AND I. G. TOLLIS, *Representations of graphs on a cylinder*, SIAM J. Discrete Math., 4 (1991), pp. 139–149.

[47] R. TAMASSIA AND J. S. VITTER, *Parallel transitive closure and point location in planar structures*, SIAM J. Comput., 20 (1991), pp. 708–725.

[48] C. THOMASSEN, *Planar acyclic oriented graphs*, Order, 5 (1989), pp. 349–361.

[49] J. VALDES, R. E. TARJAN, AND E. L. LAWLER, *The recognition of series parallel digraphs*, SIAM J. Comput., 11 (1982), pp. 298–313.

[50] S. WIMER, I. KOREN, AND I. CEDERBAUM, *Floorplans, planar graphs, and layouts*, IEEE Trans. Circuits Systems I Fund. Theory Appl., 35 (1988), pp. 267–278.

[51] S. K. WISMATH, *Characterizing bar line-of-sight graphs*, in Proc. ACM Symp. on Computational Geometry, 1985, pp. 147–152.

[52] S. K. WISMATH AND D. G. KIRKPATRICK, *Weighted visibility graphs of bar and related flow problems*, Algorithms and Data Structures (Proc. WADS'89), 1989, pp. 325–334.

SIAM J. COMPUT.
Vol. 24, No. 5, pp. 1002–1017, October 1995

004

FLOW IN PLANAR GRAPHS WITH MULTIPLE SOURCES AND SINKS*

GARY L. MILLER† AND JOSEPH (SEFFI) NAOR‡

Abstract. The problem of maximum flow in planar graphs has always been investigated under the assumption that there is only one source and one sink. Here we consider the case where there are many sources and sinks (single commodity) in a directed planar graph. An algorithm for the case when the demands of the sources and sinks are fixed and given in advance is presented. The algorithm can be implemented efficiently sequentially and in parallel, and its complexity is dominated by the complexity of computing all shortest paths from a single source in a planar graph. If the demands are not known, an algorithm for computing the maximum flow is presented for the case where the number of faces that contain sources and sinks is bounded by a slowly growing function. Our result places the problem of computing a perfect matching in a planar bipartite graph in NC and improves a previous parallel algorithm for the case of a single source, single sink in a planar directed (and undirected) graph, both in terms of processor bounds and its simple presentation.

Key words. planar graphs, flow, circulation

AMS subject classifications. 68R10, 05C38, 90B10, 90C35, 90C27

1. Introduction.

In the common formulation of the maximum flow problem, the maximum flow from a distinguished vertex in the graph, called the *source*, to another distinguished vertex in the graph, called the *sink*, is computed. Here we assume that the underlying network is planar; this case was extensively studied and more efficient algorithms were developed for it (see §2), yet the assumption was always that there is only one source and one sink.

In this paper we investigate the following problem: given a planar network with *many* sources and sinks, compute the maximum flow from the sources to the sinks. Ford and Fulkerson [3] reduced the multiple source, multiple sink problem to the single source, single sink problem by connecting the sources to a supersource and the sinks to a supersink, and then computing the maximum flow from the supersource to the supersink. In planar graphs, this reduction may *destroy* the planarity of the graph if the sources or sinks belong to different faces. Nevertheless, we would like to take advantage of the planarity of the graph to design more efficient algorithms, *sequential as well as parallel*, in the case of multiple sources and sinks.

We feel that the reformulation of the problem is more natural within the context of planar graphs and has motivation in both sequential and parallel computation. The only other attempt known to us that copes with multiple sources and sinks is by Megiddo [22], [23], whose algorithm computes (in a general graph) optimal flows, i.e., flows that are "fairly" distributed among the sources and sinks.

Maximum flow in a general network was shown to be P-complete [10], and hence it is widely believed not to have an efficient parallel algorithm. On the other hand, maximum flow can be reduced to maximum matching and this reduction implies an RNC algorithm when the edge capacities are represented in unary [16], [24]. This emphasizes the importance of solving the problem in the case of a planar network with arbitrary capacities. In the restricted case

*Received by the editors March 2, 1989; accepted for publication (in revised form) April 7, 1994. An extended abstract describing these results appeared in the Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science, 1989, pp. 112–117.

†School of Computer Science, Carnegie–Mellon University, Pittsburgh, Pennsylvania 15213-3890.

‡Computer Science Department, Technion, Haifa 32000, Israel. This research was supported by the fund for the promotion of research at the Technion, Office of Naval Research contract ONR N00014-88-K-0166 and the Computer Science Department, University of Southern California, Los Angeles, California and supported in part by National Science Foundation grant DCR-8713489. Most of this work was done while the author was at the Computer Science Department, Stanford University, Stanford, California.

of a single source, single sink, there do exist NC algorithms in both directed and undirected graphs [6], [14].

The first problem we consider is the case where the amount of flow (demand) at each source and sink is given as input, and the objective is to compute a feasible flow function. We present an efficient algorithm for this problem. The sequential complexity of our algorithm is $O(n^{1.5})$ time; the parallel complexity is $O(I(n)\log^2 n)$ time using $O(n^{1.5})$ processors where $I(n)$ is the time of computing the sum of n values. $I(n)$ can be implemented in $O(1)$ time in the concurrent read concurrent write parallel random access machine (CRCW PRAM) model and $O(\log n)$ time in the exclusive read exclusive write parallel random access machine (EREW PRAM) model. The main idea in computing the flow function in this case is redirecting the flow through a spanning tree from the sinks back to the sources. The problem then reduces to that of computing a circulation in a network with both lower and upper bounds on the capacity of the edges. Similar ideas for redirecting the flow back from the sink to the source have appeared in [5], [6], and [14] for computing the flow in the case of a single source and sink.

Note that planar graphs are different from general graphs, where it was observed that knowing the value of the maximum flow does not improve the complexity of computing the flow function [29]. In contrast to that, all the known algorithms for planar flow do take advantage of the value of the maximum flow.

We consider the special case where the sources and sinks are all on one face and the demands unknown. We present an efficient algorithm that computes the maximum flow in this case by employing a nontrivial divide-and-conquer. The sequential running time of the algorithm is $O(n\log^{1.5} n)$ time. In parallel, it can be computed using $O(n^{1.5})$ processors, where the time complexity is $O(I(n)\log^3 n)$. ($I(n)$ is the same function as previously defined.) We then show how to extend this algorithm to the case where the number of faces that contain sources and sinks is bounded by a slowly growing function.

Unfortunately, the most general problem, where the sources and sinks belong to an arbitrary number of faces and the demands are unknown, is still open; i.e., the best sequential algorithm for this problem is obtained by connecting the sources and sinks to a supersource and supersink; in parallel, there is no NC algorithm known for this problem.

An example where multiple sources and sinks are useful is the case of computing a perfect matching of a planar bipartite graph. In the standard reduction from matching to flow (see, e.g., [2]), one part of the graph is connected to a source and the other part to a sink. In general, this reduction will result in a nonplanar graph but can be utilized within our context (the demand of each source and sink is exactly one unit). This places the problem of computing a perfect matching in a planar bipartite graph in NC.

The situation in computing a perfect matching in planar graphs is very intriguing. Kasteleyn [17] had already shown how to *count* the number of perfect matchings in a planar graph, a problem that is # P-complete in general graphs, and his methods can be implemented in NC (see, e.g., [32]) as well. Yet computing a perfect matching in NC in a planar graph remains an open problem. This situation is interesting as it contradicts the current view of the computational difficulty of counting the number of solutions versus finding a solution in combinatorial problems.

Johnson [14] showed how to compute in parallel the maximum flow for the case of a single source, single sink in a directed graph. We present an algorithm for this case which improves on the number of processors and is also very *simple* in comparison with the fairly complicated algorithm of [14]. Johnson's approach [14] was the first to find the minimum cut, and then compute the flow function. Our approach is different; using parametric methods, we can find the *value* of the maximum flow; then the computation of the flow function follows easily.

Subsequent to this work, Khuller and Naor [18] considered the problem of computing a flow function in a planar graph where there are capacity constraints on both edges and vertices. Efficient algorithms for this problem are presented in [18].

The paper is organized as follows: In §2, we describe previous results in planar flow and in §3 we provide certain preliminaries. In §4, we show how to compute a circulation when edge capacities may have nonzero lower bounds. In §5, we present three applications: (i) computing the flow function when there are many sources and sinks and their demands are known; (ii) computing a perfect matching in a bipartite planar graph; (iii) improving previous algorithms for the case of a single source and sink. In §6, we show how to find the maximum flow in the case where all the sources and sinks are on the same face but the demands are not known. In §7 we extend these results to the case where the number of faces containing sources and sinks is bounded.

2. Previous results in planar flow. All the results referred to in this section deal *exclusively* with the single source, single sink maximum flow problem. Ford and Fulkerson [3] had already observed that a minimum cut in a planar graph is equivalent to a minimum weight cycle that separates the source from the sink in the dual graph. They gave an $O(n \log n)$ time algorithm to compute the minimum cut when the source and sink belong to the same face. Berge and Ghouila-Houri [1] suggested an $O(n^2)$ algorithm for computing the flow function, which is called the "uppermost path algorithm." This algorithm was implemented in $O(n \log n)$ time by Itai and Shiloach [12]. Hassin [5] gave an elegant algorithm for computing the flow function and his algorithm can be implemented in $O(n\sqrt{\log n})$ time using the method of [4] for computing shortest paths in planar graphs.

Itai and Shiloach [12] also gave an algorithm to compute the maximum flow in an undirected graph when the source and sink do not necessarily belong to the same face. Its running time was $O(n^2 \log n)$. This was improved by Reif [30] who gave an $O(n \log^2 n)$ time algorithm for computing the minimum cut in an undirected planar graph. Only Hassin and Johnson [6] completed the picture by giving an $O(n \log^2 n)$ time algorithm for computing the maximum flow in an undirected graph by generalizing the ideas of [5] and [30]. (The running time of their algorithm can be improved to $O(n \log n)$ through the methods of [4] for computing shortest paths in a planar graph.)

Computing the maximum flow in planar directed graphs is more difficult as it is not clear how to reduce the problem of computing a minimum weight cycle to that of computing a minimum weight path. Johnson and Venkatesan [15] gave an $O(n^{1.5} \log n)$ time algorithm to compute both a minimum cut and a maximal flow.

In the course of the evolution of efficient algorithms for planar flow, an interesting phenomenon occurred. The computational difficulty alternated between searching for the minimum cut on the one hand, and computing the flow function when the minimum cut is known on the other hand.

It is easy to implement the algorithm of [6] for undirected graphs in parallel, and its complexity is $O(\log^2 n)$ time using $O(n^3)$ processors. (The details are given in [14].) It should be mentioned that an alternative algorithm for computing the minimum cut in parallel in an undirected graph was given in [13].

As for directed planar graphs, an algorithm that first computes the minimum cut, and then the flow function, was given by Johnson [14]. Its complexity is $O(\log^3 n)$ time using $O(n^4)$ processors or $O(\log^2 n)$ time using $O(n^6)$ processors. The processor bounds of the algorithms of [6], [14] can be improved through the methods of [26]–[28].

3. Terminology and preliminaries. Throughout the paper let $G = (V, E)$ be an embedded planar graph where V is the *vertex set* and E is the *arc set*. An *edge* consists of two

arcs, an arc and its reflection. Let $R(e)$ be the permutation which takes an arc e to its reflection. The graphs considered may have multiple edges but every arc will have a distinct reflection.

A graph is said to be embedded in the plane if it is intuitively "drawn" in the plane with no crossing edges, where an edge and its reflection are drawn on top of each other. This definition is not very algorithmic, hence we assume that the graph is given by one of the many combinatorial definitions of planar embedding; see [25], for example. An embedding of a planar graph can be computed sequentially in linear time [11] and in parallel in $O(\log^2 n)$ time using a linear number of processors [19]. An embedding is needed to compute the dual graph. In all of our algorithms, computing an embedding will never dominate the cost of the algorithm.

An embedded graph G partitions the plane into connected regions called *faces*. Let $G^* = (F, E^*)$ be the *dual graph* of G, where F is the set of faces of G, and E^* is the set of dual arcs. There is a one-to-one correspondence between E and E^* as follows: for each arc $e \in E$, let e^* be the corresponding *dual arc* connecting the right face bordering e with the left face bordering on e.

We use a left-hand rule: if the thumb points in the direction of e, then the index finger points in the direction of e^*. The dual G^* is also known as the *clockwise dual* of G. For a vertex v, in(v) (out(v)) denotes the incoming (outgoing) set of arcs into (from) v. For an arc e, tail(e) (head(e)) denotes the vertex at the tail (head) of e.

The dual graph is planar too, but may contain self loops and multiple edges. We sometimes refer to the graph G as the *primal* graph.

A *cycle* C is an ordered set of arcs $e_0, e_1, \ldots, e_k$ such that for every $0 \leq i \leq k$, head(e_i) = tail(e_{i+1}) (mod $k+1$). The cycle C is simple if the vertices between the arcs are distinct. Thus all cycles are *directed*.

3.1. Flow in planar graphs. In this section we formally define the planar flow problem. We generalize the problem in two ways. First, we allow multiple sources and sinks. Second, we introduce vertices with fixed flow demands. This is the definition we shall use throughout the rest of this paper.

DEFINITION 3.1. *A flow graph with sources, sinks, capacities, and demands is the following five-tuple* (G, S, T, c, d) *such that*

- $G = (V, E)$ *is a graph, where* V *is a set of vertices and* E *is a set of arcs;*
- *the sources and sinks with variable demands are* $S \subseteq V$ *and* $T \subseteq V$, *respectively, where the sets* S *and* T *are disjoint;*
- *the map* $c : E \to \Re$ *is the edge capacity (for all* $e \in E$: $c(e) = -c(R(e))$*);*
- *the map* $d : V - \{S, T\} \to \Re$ *is the demand at nonsource and nonsink vertices, i.e., vertices for which the demand is fixed.*

Observe, that $c(e)$ and $d(v)$ may be negative. Vertices for which $d(v) > 0$ are called sources with fixed demands, and vertices for which $d(v) < 0$ are called sinks with fixed demands.

DEFINITION 3.2. *A function* $f : E \to \Re$ *is a flow function if*

(i) $\forall e \in E : f(e) = -f(R(e))$;

(ii) $\forall v \in V - \{S, T\} : \sum_{\text{tail(e)=v}} f(e) = d(v)$.

DEFINITION 3.3. *The function* f *is a legal (or feasible) flow function if, in addition,* $\forall e \in E$, $f(e) \leq c(e)$, *where* $c(e)$ *denotes the capacity of edge* e.

Given a flow function f, we define for all $v \in V$, $f(v) = \sum_{\text{tail(e)=v}} f(e)$.

In the maximum flow problem, we are looking for a legal flow that maximizes the *total* amount of flow entering T (or leaving S). The amount of flow entering the sinks is also called the *value* of the flow. A *circulation graph* is a flow graph with no sinks or sources, i.e., $S = T = \emptyset$ and $d(v) = 0$ for all vertices. A *circulation function* is a flow function with

respect to a circulation graph. A flow graph with *fixed demands* is a flow graph with no sinks or sources that have variable demand, i.e., $S = T = \emptyset$.

Flow graphs and flow functions can be added and subtracted as follows: given a flow graph G, flow functions f_1 and f_2, and a real number α, the sum $f = \alpha f_1 \pm f_2$ is the flow where, for every edge e, $f(e) = \alpha f_1(e) \pm f_2(e)$. We define the addition of a flow graph and a flow function to obtain a new flow graph. Let (G, c, d) be a flow graph with capacity c and demands d, and let f be a flow function defined on G. The flow graph $G + f$ will be the triple $(G, c + f, d + f)$. The demand function $d + f$ is only defined for vertices v for which d is defined and $(d + f)(v) = d(v) + f(v)$. The *residual graph* with respect to a given flow f is formally defined as $G - f$. Observe that the residual graph with respect to a legal flow function cannot have negative capacities.

An *augmenting path* in a flow graph G is a path that starts at a source, ends at a sink, and uses only arcs with strictly positive capacity. We do not distinguish between an augmenting path as a set of arcs or, alternatively, as a flow of one unit on the arcs in the augmenting path. We say this more formally. A path P is *edge disjoint* if the arcs in P are distinct and no arc and its reflection both belong to P. Let P be an edge disjoint path from a source to a sink. The *unit flow* on P is defined as follows:

$$P(e) = \begin{cases} 1 & \text{if } e \in P, \\ -1 & \text{if } e \in R(P), \\ 0 & \text{otherwise.} \end{cases}$$

A *potential graph* is a graph where each arc is assigned a weight $w \in \Re$. A *potential function* on a potential graph $G = (V, E)$ is any real valued function p defined on the vertices. The function p is called *consistent* if the potential difference over each edge is not larger than the edge weight, i.e., $\forall e \in E, \quad w(e) \geq p(\text{head}(e)) - p(\text{tail}(e))$.

If $G = (V, E)$ is an embedded planar flow graph then its *potential dual graph* is $G^* = (F, E^*)$ such that $w(e*) = c(e)$. Given a consistent potential function p defined on G^*, we obtain a flow function f by setting $f(e) = p(\text{head}(e^*)) - p(\text{tail}(e^*))$. Clearly, f satisfies condition (i) of Definition 3.2. To see that it also satisfies condition (ii) in the case where all the demands are zero, we use the following easy yet fundamental proposition proved in [5] and [14].

PROPOSITION 3.1. *Let $C = e_1^*, \ldots, e_k^*$ be a cycle in the dual graph. Then the flow f satisfies the following property: $f(e_1) + \cdots + f(e_k) = 0$.*

Let the cycle C in the dual graph correspond to the set of primal edges adjacent to a primal vertex. Then it follows that f satisfies condition (ii) of Definition 3.2. Hence, any potential function induces a circulation in a planar flow graph. Furthermore, if the potential function is consistent then the flow function is legal.

The use of a potential function as a mean of computing a flow (and circulation) was first suggested by Hassin [5] and was later elaborated by [6] and [14], but not stated in terms of circulations.

An important procedure that will be used in all of our algorithms is computing all shortest paths from a given vertex in a planar graph which may contain negative edge weights. The sequential complexity of this procedure is $O(n^{1.5})$ using the generalized nested dissection method of [20]. To compute in parallel shortest distances, the parallel nested dissection implementation of [26]–[28] requires $O(n^{1.5})$ processors; the time complexity is $O(I(n) \log^2 n)$, where $I(n)$ is the parallel time of computing the sum of n values. $I(n)$ can be implemented in $O(1)$ time in the CRCW PRAM model and $O(\log n)$ time in the EREW PRAM model.

To implement the method of nested dissection we need to compute small separators in planar graphs. A small separator can be computed sequentially in linear time; see [25]. In parallel, Gazit and Miller [7], [8] provided a procedure for computing small separators, where

the complexity is $O(n^{1+\epsilon})$ processors and the running time is dominated by the running time of a procedure for computing a maximal independent set in a graph (not necessarily planar). The current best bound is $O(\log^3 n)$ time using a linear number of processors [9].

4. Computing circulations. In this section we show how to compute a circulation in a planar graph. It is clear that the difficult case occurs when some of the edges have negative capacity. Otherwise, we can set the flow on each edge to be zero and obtain a legal circulation. An edge which has negative capacity may be viewed as having a lower bound on the flow through it; for example, an edge which has a lower bound of a and an upper bound of b on the capacity can be replaced by two edges of opposite direction, which have capacities b and $-a$. We provide a precise characterization of planar circulation problems that have feasible solutions.

As stated in the previous section, the key idea is to compute a consistent potential function on the faces of the planar graph and define the flow in each edge as the potential difference of the two faces that border the edge. We show that finding such a potential function reduces to computing the shortest paths from a single source. Throughout this section, let G denote a planar circulation graph with capacities $c(e)$ and a dual G^* which has weights $w(e)$.

A *shortest path* numbering for a graph (from some source vertex x) is an assignment of real values to the vertices such that the value of a vertex y equals the minimum weight path from x to y. We shall simply call it an SP numbering. It is well known that every strongly connected graph has an SP numbering if and only if it has no negative weight cycles.

LEMMA 4.1. *Let G be a connected, embedded circulation graph. Then, the following conditions are equivalent:*

1. *G^* has no negative weight cycles.*
2. *G^* has an SP numbering from every vertex (face in G).*
3. *G has a legal circulation.*

Proof. By the comment above, if G^* does not contain negative weight cycles, then G^* has an SP numbering from every vertex. To see that the existence of an SP numbering also implies the existence of a circulation, let the potentials assigned to the faces be equal to their SP numbering from some arbitrary vertex. Let p denote the potential function and e^* be a dual edge directed from face F to face F'. Since p is an SP numbering, it follows that

$$p(F') - p(F) \leq w(e^*).$$

Therefore, p is a consistent potential function and G has a legal circulation by Proposition 3.1.

To see that if G has a legal circulation, then G^* has no negative weight cycles, assume the contrary: G^* contains a negative weight cycle $e_1^*, \ldots, e_k^*$ yet G has a legal circulation f. Note that a cycle in the dual graph G^* separates the plane into two regions. Hence the incoming flow into it has to be equal to the outgoing flow, i.e., $f(e_1) + \cdots + f(e_k) = 0$, but $f(e_1) + \cdots + f(e_k) \leq c(e_1) + \cdots + c(e_k) < 0$, resulting in a contradiction. □

This lemma gives a precise characterization for the existence of a feasible circulation in a planar graph: the dual graph cannot contain negative weight cycles.

To summarize, the potential function is computed as follows: Choose an arbitrary vertex in the dual graph and compute the shortest path from it to all other vertices. The length of the shortest path is defined as the potential of the vertex. The flow on each edge is defined as the potential difference of the two faces bordering it.

The cost of computing a legal circulation is dominated by the cost of computing one SP numbering. The sequential and parallel complexity of computing an SP numbering are given in §3.1. Thus, we have the following theorem.

THEOREM 4.1. *A legal circulation can be computed in a planar graph in $O(n^{1.5})$ time sequentially. In parallel it can be computed using $O(n^{1.5})$ processors; the time complexity is $O(I(n)\log^2 n)$, where $I(n)$ is the parallel time of computing the sum of n values. $I(n)$ can be implemented in $O(1)$ time in the CRCW PRAM model and $O(\log n)$ time in the EREW PRAM model.*

5. Applications of the circulation algorithm. In this section we present three applications of the circulation algorithm. In §5.1 we show how to compute a feasible flow function in a flow graph without variable sources and sinks, i.e., when all demands are fixed and known in advance. In §5.2 we show that a perfect matching in a planar bipartite graph can be computed in NC. In §5.3 we present a simple algorithm for computing the maximum flow for the case of a single source and sink with variable demand.

5.1. Computing the flow function for fixed demands. In this section we assume that our input is a planar graph with many sources and sinks, where the demand at each source and sink is known in advance. We present an efficient algorithm that determines whether a feasible solution, i.e., a solution that satisfies the demands, exists, and if so computes it. Suppose that a demand function d is defined on the vertices (see §3.1) such that sources have positive demands and sinks have negative demands. The rest of the vertices have zero demand. We may assume that the sum of the demands is zero since there is no feasible solution otherwise.

The main idea of the algorithm is that computing a flow function with fixed demands can be reduced to computing a circulation. This reduction is achieved via a tree T that spans the sources and sinks. We add new edges to the graph parallel to the edges of T, resulting in a new graph G'.

Recall that in the algorithms of [6] and [14], a similar reduction is achieved by returning the flow from the sink to the source via a simple path. This idea is generalized here and the spanning tree T is used to redirect the flow from the sinks back to the sources. Any circulation computed in G' will induce a flow satisfying the demands in G.

SKETCH OF ALGORITHM (I).
Input: a planar flow graph (G, c, d) where the demand function is defined for all vertices of G.
Output: a legal flow function f that satisfies the demands.

1. Find any flow function f' for G.
2. Construct the residual flow graph $G' = G - f'$ where G' is a circulation graph.
3. Compute a circulation f'' in G'.
4. Return the flow $f = f' + f''$.

We now elaborate on the implementation of each step. To do step 1, we first compute a spanning tree T in G, where an edge in T consists of both an arc and its reflection. We now describe how the spanning tree is used to redirect the flow from the sinks back to the sources.

For all arcs that are not in T, we set f' to zero. An arc $e \in T$ separates the tree into two subtrees, called tail and head. T_{tail} is the subtree adjacent to the tail of e and T_{head} is adjacent to the head of e. Let $f'(e)$ be defined as $\sum_{v \in T_{\text{tail}}} d(v)$. The last term is also equal to $-\sum_{v \in T_{\text{head}}} d(v)$ since T is a spanning tree and the sum of the demands on all vertices in the graph is zero. To see that $f'(v) = d(v)$, let E' be the set of arcs whose head is v. Note that if an arc $e \in E'$ is not in T, then $f'(e) = 0$.

$$f'(v) = \sum_{e \in E'} f'(e) = \sum_{w \in V - \{v\}} -d(w) = d(v).$$

We need only show that $f' + f''$ is a legal flow which meets the demands, given that f'' is a circulation. The flow $f' + f''$ meets the demand since

$$(f' + f'')(v) = f'(v) + f''(v) = f'(v) + 0 = d(v).$$

To see that it is legal,

$$(f' + f'')(e) = f'(e) + f''(e) \leq f'(e) + (c - f')(e) = c(e).$$

The most expensive part of the algorithm is step 3, where all shortest paths from a vertex in the dual graph are computed. The sequential and parallel complexity of computing all shortest paths from a given vertex in a graph with negative edge weights is discussed in §3.1. Thus, we have the following theorem.

THEOREM 5.1. *A flow function with fixed demands can be computed in a planar graph in $O(n^{1.5})$ time sequentially. In parallel, it can be computed using $O(n^{1.5})$ processors; the time complexity is $O(I(n)\log^2 n)$, where $I(n)$ is the parallel time of computing the sum of n values. $I(n)$ can be implemented in $O(1)$ time in the CRCW PRAM model and in $O(\log n)$ time in the EREW PRAM model.*

5.2. Finding a perfect matching. In this section we show how to compute a perfect matching in a planar bipartite graph $G = (A, B, E)$, where A and B are the two parts of the vertex set. In the standard reduction from matching to flow (see, e.g., [2]), E is directed from A to B, a source s is connected to all the vertices of A, and a sink t is connected to all the vertices of B. All the edges in the reduced graph have unit capacity; the saturated edges in a maximum flow constitute a maximum matching in G. Obviously, this reduction may, in general, destroy the planarity of the graph.

To compute the perfect matching efficiently, each vertex in A becomes a source and each vertex in B becomes a sink. The demand at each source and sink is exactly one unit. The edges are oriented as before from A to B with unit capacity.

The sequential complexity of our algorithm is $O(n^{1.5})$ time and it matches the best sequential bound for computing a maximum matching in a planar graph [21]. In parallel, our result places in NC the problem of computing a perfect matching in a planar bipartite graph.

5.3. Planar maximum flow with a single source and sink. In this section we show how to improve the algorithm of [14] in the case of a single source, single sink in a directed planar graph. We present a simple algorithm for this problem and improve the processor bound with respect to [14]. We also handle the case where the capacities are possibly negative and the case where some of the vertices may have fixed demands by returning flow as in §5.1, Algorithm (I).

The approach taken in [14] is first to find the minimum cut and then compute the flow function. We proceed differently; to apply Algorithm (I), we need to compute the *value* of the minimum cut, denoted by α. This will be done by a parametric search method. Once the value of the minimum cut is known, the flow function can be computed by Algorithm (I).

Recall that in Algorithm (I) the flow is returned from the sinks to the sources via a spanning tree. Note that in the case of a single source and sink, the spanning tree is a simple path, denoted by P, from t to s. The difficulty is that we do not know how much flow to return from t to s. We first guess an initial value α, which is greater than or equal to the value of the maximum flow from s to t; e.g., initially, we set α equal to the sum of the capacities of the edges leaving the source. Throughout the rest of this section, we let f_α denote the flow, which is α for arcs in P, $-\alpha$ for arcs in $R(P)$, and zero otherwise. Let $G' = G - f_\alpha$.

The characterization of feasible circulations in Lemma 4.1 implies that we should compute the maximum α (if it exists at all) such that the dual graph of G' does not contain negative

weight cycles. If our initial guess was too large, then in the dual graph of G', where the shortest paths are computed, there must be a negative weight cycle. (Otherwise, a circulation that would correspond to a flow whose value is α can be computed.) It should be noted that there may be negative cycles in the dual graph that exist independently of the value we set for α. This can only happen if there is no feasible flow in G even if α is set to zero. If this case this is detected we halt, returning with no flow. We can henceforth assume that there exists a legal flow function for G for some positive value of α.

Let C be a simple cycle in the dual graph G^*. The *net crossings* of the path P by the cycle C is defined as being equal to the number of times C crosses P from right to left minus the number of crossings from left to right. (Right and left are defined with respect to the partition of the plane by the path P.)

We observe the following straightforward fact.

LEMMA 5.1. *The net crossings of P by C is either* 0 *or* ± 1.

Proof. Assume the cycle C crosses the path P more than once. We prove that two consecutive crossings on P alternate between left-right crossings and right-left ones. Suppose, to the contrary, that edges $e, e' \in C$ are two consecutive crossings that are oriented in the same direction, where e' is to the right of e with respect to their orientation. The cycle C partitions the plane into two regions, interior and exterior, and without loss of generality let the interior be the region left of the cycle C with respect to its orientation. Now, however, there are two points in the plane, one point to the right of e (in the exterior) and the other to the left of e' (in the interior), that can be connected without crossing the cycle C. Hence, there can be at most one crossing that is not "canceled" and the correctness of the lemma follows. □

The following definitions hold for both sequential and parallel algorithms. An algorithm that computes shortest paths in a graph is called *oblivious* if any decision on which paths in the graph to compare is independent of the weights of the edges. In particular, for a fixed (unweighted) graph, an oblivious algorithm will always compare the same paths for any assignment of weights to the edges. An algorithm that computes shortest paths in a graph is called *additive* if, for any nonsimple cycle C, its weight is computed in a time step subsequent to the time steps in which the weight of each of its componenets that form simple cycles is computed.

We first describe a generic algorithm for finding the maximum feasible α under the assumption that an (sequential or parallel) algorithm $\mathcal{A}$ for computing shortest paths in a graph that is both oblivious and additive is available.

GENERIC ALGORITHM.

Input: a planar flow graph (G, c, d) where the demand function is a variable for precisely one source and one sink; an oblivious and additive algorithm $\mathcal{A}$ for computing shortest paths in G.
Output: the maximum legal flow.

1. Find any flow f' for G; **Set** $G \leftarrow G - f'$. (Flow f' satisfies the fixed demands in the graph.)
2. Find a simple path P from s to t and construct flow f_α for some large constant α.
3. Set $G_\alpha \leftarrow G - f_\alpha$.
4. Test whether G_α^* has negative cycles. Run Algorithm $\mathcal{A}$ on G_α^*:
   (a) Let τ be the first step in which a negative cycle is detected (in Algorithm $\mathcal{A}$) and let l be the weight of the most negative cycle detected at Step τ.
   (b) Set $\alpha \leftarrow \alpha + l$; restart Algorithm $\mathcal{A}$, i.e., goto Step 3.

THEOREM 5.2. *Assume that the running time of Algorithm $\mathcal{A}$ is $O(T)$. Then, the generic algorithm terminates after at most $O(T^2)$ steps and computes the maximum feasible value of α.*

Proof. Let τ denote the first step of Algorithm $\mathcal{A}$ in which a negative cycle is detected. We will prove that after updating the value of α and restarting Algorithm $\mathcal{A}$, a negative cycle can be detected only in time steps subsequent to τ. Hence, the generic algorithm terminates after at most $\sum_{\tau=1}^{T} \tau$ steps, which is at most $O(T^2)$ steps.

Let $\mathcal{C}$ denote the set of cycles in which Algorithm $\mathcal{A}$ has computed their weight up to Step τ. We claim that after updating the value of α, the weight of all cycles in the set $\mathcal{C}$ must be nonnegative. Since Algorithm $\mathcal{A}$ is oblivious, it follows that in any subsequent iteration, a negative cycle cannot be detected before Step $\tau + 1$.

To prove the claim, suppose to the contrary that in time step $\tau' \leq \tau$, a negative weight cycle $C \in \mathcal{C}$ is discovered in $G^*_{\alpha+l}$. (Assume that τ' is the first step in which a negative cycle is detected.) We first prove that C must be a simple cycle. If C is not a simple cycle, then it can be decomposed into a set of simple cycles. By the additivity property of Algorithm $\mathcal{A}$, this set of simple cycles must belong to $\mathcal{C}$. Since the weight of C is the sum of the weights of the simple cycles in its decomposition, some of the simple cycles must have negative weight. Again, by the additivity property, the weight of these simple cycles will be computed in a time step τ'', where $\tau'' < \tau'$. Therefore, we can assume that C is a simple cycle.

Recall that for every negative weight simple cycle in $\mathcal{C}$, the net number of crossings of P must be 0 or ± 1. Therefore, the weight of cycle C in G^*_α cannot be less than l, implying that all the simple cycles in $G^*_{\alpha+l}$ belonging to $\mathcal{C}$ cannot be of negative weight. In step 4(b) the value of α can only decrease, and hence the weight of the cycles in $\mathcal{C}$ will remain nonnegative.

It remains to prove that the algorithm computes the maximum α. Let $\alpha_{\max}$ denote the final value of α computed by the algorithm and let C be the most negative cycle detected by the algorithm in the last step in which a negative cycle was detected. It is easy to see that in any graph $G_{\alpha'}$ (where $\alpha' > \alpha_{\max}$) the weight of cycle C must be negative. □

5.3.1. Implementing the generic algorithm. The most efficient way of implementing the generic algorithm would be by using the nested dissection method of [20] and its implementation in parallel by Pan and Reif [26]–[28]. We provide a high level description of it.

The basic idea underlying the method of nested dissection is partitioning the graph by a family of separators, where each separator partitions the graph into equal size components (up to constant factors). Each separator is a cycle, and and we refer to the two components of the graph generated by the cycle as being "inside" and "outside" the separator. Hence we can think of the separators as forming a binary tree as follows: the root of the tree is the graph G; the descendants of each vertex in the tree are the two components generated by the separator. The shortest distances in the graph are computed bottom up in the tree.

Pan and Reif obtained their best time bounds [28] by streamlining the computation. Intuitively, this can be thought of as if vertices in a certain level in the tree of separators begin computing their transitive closure before the vertices in the levels below them have finished computing their transitive closure.

For us, the important feature of shortest path algorithms based on nested dissection is that they are oblivious and additive.

THEOREM 5.3. *The maximum flow in a directed planar graph with a single source, single sink can be computed in $O(\log^4 n)$ time using $O(n^{1.5})$ processors in the CRCW model.*

Proof. The proof follows from the discussion. □

The sequential running time was shown in [15] to be $O(n^{1.5} \log n)$.

Another oblivious and additive algorithm for computing shortest paths is via matrix multiplication and doubling-up. The computation of the shortest paths proceeds in this case by successive squaring of the adjacency matrix A of G^* until we get A^n. Let k be the first iteration in which a negative entry appears in the diagonal of A^{2^k} and let l be the most negative entry of the diagonal in that iteration. We update α by l, i.e., setting $G' \leftarrow G - f_{\alpha+l}$, and start the

computation of the shortest paths from the beginning. It follows from the proof of Theorem 5.2 that A^{2^k} will not have negative entries any more in its diagonal. Hence, at most $\log n$ computations of the shortest paths algorithm suffice to compute the value of the minimum cut.

6. Maximum flow on the disk. In this section we describe an algorithm for computing a maximum flow for the case where all the sources and sinks with variable demands lie on the same face. Without loss of generality, one can assume that the sources and sinks are on the outer face and that they alternate, i.e., there are no two consecutive sources or sinks. These two properties will be maintained during the recursive calls to the algorithm. We discuss two cases. In the first case we will assume that G has no negative capacities and no vertices with nonzero demands, i.e., the zero flow is a legal flow. In the second case, we allow negative capacities and nonzero demands and we show how to reduce this case to the first case.

6.1. Maximum flow on the disk with positive capacities and zero demands. Let G be a flow graph and f be a maximum flow on G. Among all minimum cuts separating the sources from the sinks, we are interested in the "first" minimum cut, defined as follows: Let W be the set of vertices that are reachable from the sources in $G - f$. The *Ford–Fulkerson cut* with respect to f is the set of edges between W and $V - W$. Suppose the sources and sinks of the outer face are separated into two consecutive sets L and R; the maximum flow f from L to R is defined as the flow that maximizes the flow from the sources in L to the sinks in R. In particular, we can set the demand of each sink in L and each source in R to zero.

The main idea of the algorithm is the following: Divide the vertices of the outer face into two (aforementioned) sets and compute the maximum flow from L to R. The Ford–Fulkerson cut associated with this flow decomposes the disk into regions and in each region the maximum flow is computed recursively. In the last step of the algorithm, the maximum flow is computed from R to L. We prove that when the algorithm terminates, the flow cannot be augmented.

Assume that after the flow is computed from L to R, the edges of the Ford–Fulkerson cut are removed from the graph in the recursive calls.

Let V_s denote the sources and sinks belonging to a set of vertices V. Let $\mathcal{C}$ denote the set of connected components of $G - f$ after the edges of the Ford–Fulkerson cut are deleted.

We are now ready to present an outline of the algorithm for computing the maximum flow.

SKETCH OF ALGORITHM (II).
Input: a planar flow graph (G, S, T, c, d) where the demand function is zero for all vertices of G, the capacities are all positive, and the sources and sinks are on the outer face.
Output: the maximum flow function f in G.
If G has at most one source or one sink then return the zero flow.
Else:

1. Divide the sources and sinks into two consecutive sets, L and R, such that $|L_s| = |R_s|$ and L contains at least as many sources as R.
2. Compute a maximum flow f_{LR} from L to R. Compute the residual graph $G' = G - f_{LR}$.
3. Delete the edges of the Ford–Fulkerson cut from G' and compute $\mathcal{C}$, the connected components of G'; recursively, compute the maximum flow for each component $c \in \mathcal{C}$. The bottom of the recursion is when a component c contains a unique source and no sinks, or vice versa. Let $f_{L\&R}$ be the sum of the flows computed for each component.
4. Compute the residual graph $G'' = G - f_{LR} - f_{L\&R}$. Compute the maximum flow f_{RL} from R to L in G''.
5. Return the flow $f_{LR} + f_{L\&R} + f_{RL}$.

We now elaborate on the steps of the algorithm. In step 2 connect all the sources in L to a new vertex, a supersource, and all the sinks in R to a new vertex, a supersink. This can be done without destroying the invariant that all the sources and sinks of G lie on the outer face, since we have set the demand of the sinks in L and the sources in R to zero. The problem then reduces to computing a maximum flow in an $\{s, t\}$ planar graph, a graph in which both the source and the sink are on the same face [3], [12], [5]. Observe that all recursive calls are for graphs with nonnegative capacities and zero demands.

In step 3 the maximum flow in each $c \in \mathcal{C}$ is recursively computed. The capacities of the edges in c are the residual capacities with respect to the flow computed in step 2. We now recursively compute the maximum flow inside c. If a connected component contains vertices from both L_s and R_s, then there can be only two cases: (i) sinks from L with sources and sinks from R; (ii) sources from R with sinks and sources from L. In the recursive call, in the first case we connect all the sinks that belong to L to a supersink, and in the second case we connect all the sources that belong to R to a supersource. This is done to ensure that the number of sources and sinks decreases by a constant factor in each recursive call. Observe that the flows computed in each component are disjoint and hence the sum is a legal flow.

In step 4 we compute the maximum flow from R to L similar to step 2.

We now prove the correctness of the algorithm.

We first need a technical fact about writing a flow as a sum of augmenting paths which are viewed as flows. Let G be a flow graph with sources and sinks, nonnegative capacities, and zero demands. Furthermore, let f be a legal flow for G. A sum of augmenting paths $\alpha_1 f_1 + \cdots + \alpha_k f_k = f$ is a *positive decomposition* of f if

(i) $\alpha_i > 0$ for $1 \leq i \leq k$;

(ii) the arcs e and $R(e)$ cannot both belong to any of the paths $f_1, \ldots, f_k$;

(iii) each path starts at a source and ends at a sink.

LEMMA 6.1. *Let G and f be as above. Then there exists another legal flow f' on G which has a positive decomposition into augmenting paths and agrees with f on the sinks and sources of G.*

Proof. We pick a source s in G such that $f(s) > 0$. Starting from s and ending at some sink, we pick a path P of arcs such that the flow in f on each arc is positive. Let α_1 be the minimum flow on any arc in P. Set $f = f - \alpha_1 f_1$ and observe that f is still a legal flow. We continue in this greedy fashion until all sinks have zero flow. Let $f_1, \ldots, f_k$ be the constructed augmenting paths. At this point we set $f' = \alpha_1 f_1 + \cdots + \alpha_k f_k$ and discard the remaining flow in f. □

THEOREM 6.1. *Algorithm* (II) *correctly computes a maximum flow.*

Proof. The proof is by induction on the number of sinks and sources in G. If G has only one source or sink, then its flow must be zero since the demand at all other vertices is assumed to be zero. Since Algorithm (II) returns zero in this case, we may assume inductively that at the end of step 4, a maximum flow was computed in each connected component $c \in \mathcal{C}$. That is, there is no augmenting path contained in c for the flow graph $G - f_{LR} - f_{L\&R}$. In the time analysis we shall bound more closely the number of sinks and sources in each recursive call, but it is clear that each call has strictly fewer sources and sinks.

To prove the theorem, we have to show that there are no augmenting paths from any source s to any sink t in $G - f_{LR} - f_{L\&R} - f_{RL}$. By Lemma 6.1 we can replace the flow f_{RL} with another legal flow f'_{RL} such that it can be written as the positive sum $\alpha_1 f_1 + \cdots + \alpha_k f_k$. It will suffice to show that there are no augmenting paths with respect to the flow $G - f_{LR} - f_{L\&R} - f'_{RL}$. Note that all the paths f_i $(1 \leq i \leq k)$ are augmenting paths with respect to the flow $G - f_{LR} - f_{L\&R}$. Furthermore, for any edge e belonging to the Ford–Fulkerson cut, only $R(e)$ may belong to an augmenting path f_i. Hence an augmenting path f_i can only leave a connected component but not enter a connected component.

Now suppose that A is an augmenting path from a source s to a sink t in $G - f_{LR} - f_{L\&R} - f'_{RL}$. There are four cases, depending on whether s is in L or R and t is in L and R, which we denote by LL, LR, RR, and RL. We show that the existence of A results in a contradiction in each case.

We know that A cannot be a type RL since step 4 computed a maximum flow from right to left.

Suppose that A is of type LL. We claim that one of the arcs of A must have zero residual capacity in $G - f_{LR} - f_{L\&R}$. If A contains an arc of the Ford–Fulkerson cut, the claim is clearly true. If not, then A is contained in one of the connected components c from step 3. Since step 3 returns a maximum flow for c (by the induction hypothesis), the path A cannot be augmenting for $G - f_{LR} - f_{L\&R}$ and, therefore, the arc must exist. Let e be the first such arc on A. Since e has residual capacity in $G - f_{LR} - f_{L\&R} - f'_{RL}$, it must be the case that one of the augmenting paths f_i from the positive decomposition of f'_{RL} contains the arc $R(e)$. Consider the following path A' that consists of the arcs of A up to but not including e plus the arcs in f_i following but not including $R(e)$. By the observation above, the arcs in f_i following $R(e)$ must belong to c. Thus, the path A' is an augmenting path for $G - f_{LR} - f_{L\&R}$, resulting in a contradiction. Therefore, type LL augmenting paths do not exist.

We handle the last two cases, LR and RR, together. As in case LL, some arc on A must have zero capacity in $G - f_{LR} - f_{L\&R}$. In this case let e be the last arc on A with zero capacity. Furthermore, let f_i be an augmenting path containing $R(e)$. We construct an augmenting path A' from arcs on f_i before $R(e)$ and arcs on A that follow e. It follows that A' is an augmenting path for $G - f_{LR} - f_{L\&R}$, again a contradiction. □

THEOREM 6.2. *The running time of Algorithm* (II) *is* $O(n \log^{1.5} n)$ *sequentially. In parallel, it can be computed using* $O(n^{1.5})$ *processors; the time complexity is* $O(I(n) \log^3 n)$, *where* $I(n)$ *is the parallel time of computing the sum of* n *values.* $I(n)$ *can be implemented in* $O(1)$ *time in the CRCW PRAM model and in* $O(\log n)$ *time in the EREW PRAM model.*

Proof. In steps 2 and 5 we compute the maximum flow in an $\{s, t\}$-planar graph. This can be done by Hassin's algorithm [5], and its time complexity is $O(n\sqrt{\log n})$ sequentially [5]. In parallel, Hassin's algorithm can be implemented in $O(I(n) \log^2 n)$ time and $O(n^{1.5})$ processors by using the shortest path procedure outlined in §3.1. Since all the recursive calls at a given level of the recursion are on vertex disjoint subgraphs, we will only need $O(n^{1.5})$ processors for the full algorithm.

Observe that if G has $2k$ sinks and sources, then each connected component will have at most $k + 1$ sinks and sources for k odd and k sinks and sources for n even. It follows that the number of alternations of sources and sinks in each connected component of C is reduced by a constant fraction. Note that in step 3, at most one source or one sink is added instead of the edges of the Ford–Fulkerson cut.

Let $T_n(a)$ and $P_n(a)$ denote the time and number of processors, respectively, and let n and a denote the number of vertices and alternations, respectively. For the parallel running time we get the following recursive formula:

$$T_n(a) \leq T_n(a/2 + 1) + O(I(n) \log^2 n),$$

and hence the running time of the algorithm is $O(I(n) \log^3 n)$. As already mentioned, the number of processors then needed is $O(n^{1.5})$.

For the sequential running time we get the following recursive formula:

$$T_n(a) \leq 2T_n(a/2 + 1) + O(n\sqrt{\log n}),$$

and hence the sequential running time of the algorithm is $O(n \log^{1.5} n)$. □

6.2. Maximum flow on disk with negative capacities. In this section we show how to reduce the maximum flow problem on a disk to the special case needed in §6.1. There are two conditions that must be met in order to apply Algorithm (II):

- Vertices must have zero demand.
- Edges must have nonnegative capacities.

The reduction is quite straightforward and consists of the following steps: First, choose any flow f that meets the demands. Then, find a maximum legal flow f' in $G - f$ and return the flow $f + f'$.

The first step is implemented very similarly to algorithm (I): the flow is returned from the sinks to the sources via a spanning tree. Now, all vertices which are not variable sources or sinks have demand equal to zero. However, we may still have negative capacities. To reduce the case of negative capacities to the case of nonnegative capacities we need only find any flow. We introduce a supersource-sink vertex s and connect every source and sink to s. We also set the demand at s and at every source and sink to zero. This gives a planar flow graph G, which is in fact a circulation graph. Thus, the second step above will actually consist of two substeps: first, find any flow f'' in $G + f$ using the reduction to the circulation problem; then, find a maximum flow f' in $G + f - f''$ using Algorithm (II).

7. Maximum flow for a bounded number of faces containing sources and sinks. There are several extensions of our work. As previously mentioned, efficiently computing (sequentially and in parallel) a maximum flow in a planar graph with many sources and sinks with variable demands is still open. However, we observe that we can provide efficient sequential and parallel algorithms for the case where the sources and sinks belong to a fixed or slowly growing number of faces. As in §6, we discuss the special case when all capacities are nonnegative and all demands are zero.

First observe that the following greedy algorithm computes a maximum flow with many sources and sinks. Suppose that G is any flow graph with sources $s_1, \ldots, s_k$ and sinks $t_1, \ldots, t_l$. We claim that the following algorithm finds a maximum flow.

SKETCH OF ALGORITHM (III).
Input: a planar flow graph (G, S, T, c, d) where the demand function is 0 for all vertices and all capacities are positive.
Output: a maximum flow function f.

1. Set $f = 0$
2. **For** $1 \le i \le k$ and $1 \le j \le l$ **do**
   (a) Set the demand at all s_u and all t_v for $u \ne i$ and $v \ne j$ to zero and find a maximum flow f_{ij} from s_i to t_j in $G - f$.
   (b) **Set** $f = f + f_{ij}$.

   Return f

LEMMA 7.1. *Algorithm* (III) *computes a maximum flow.*

We next observe that the proof of correctness of Algorithm (II) is actually independent of the following: (i) the fact that the sources and sinks are all on one face; (ii) the planarity of the underlying graph. This implies that the following generic algorithm computes a maximum flow in a graph G (not necessarily planar) with many sources and sinks:

1. Partition the sources and sinks into two disjoint sets L and R.
2. Compute the maximum flow from L to R, i.e., from the sources in L to the sinks in R. Let C denote the Ford–Fulkerson minimum cut with respect to the maximum flow.
3. Remove the edges of C from the graph G. Recursively compute a maximum flow in each connected component (in the residual graph).
4. Compute a maximum flow from R to L (in the residual graph).

Let G be a planar graph with variable sources and sinks that lie on at most k faces of G, denoted by $F_1, \ldots F_k$. We now show how to combine Algorithm (III) with the generic algorithm to efficiently compute a maximum flow in G.

Step 1 in the generic algorithm is implemented as follows: the sources and sinks on face F_i $(1 \leq i \leq k)$ are partitioned into two sets, L_i and R_i, in the same way that the sources and sinks on the disk are partitioned in Algorithm (II). The set L is defined as the union of the sets L_i $(1 \leq i \leq k)$ and the set R is defined as the union of the sets R_i $(1 \leq i \leq k)$.

To implement step 2, first connect the sources in each set L_i to a supersource s_i, and the sinks in each set R_i to supersink t_i. This operation does not violate the planarity of the graph. The maximum flow from L to R is computed by Algorithm (III). Computing the flow from source s_i to sink t_j in Algorithm (III) is an instance of the problem of computing the maximum flow in a directed planar graph with one source and one sink. Step 4 is implemented similarly.

Note that in step 3, the number of alternations of sources and sinks is reduced by a constant factor for each face F_i $(1 \leq i \leq k)$.

Hence we get recursive formulas for the running time which are similar to those obtained in the proof of Theorem 6.2. For the parallel running time we get

$$T_n(a) \leq T_n(a/2 + 1) + O(k^2 I^2(n) \log^4 n),$$

and hence the running time of the algorithm is $O(k^2 log^5 n)$ in the CRCW model. The number of processors needed is $O(n^{1.5})$.

For the sequential running time we get

$$T_n(a) \leq 2T_n(a/2 + 1) + O(k^2 n^{1.5} \log n),$$

and hence the sequential running time of the algorithm is $O(k^2 n^{1.5} \log^2 n)$.

We conclude with the following theorem.

THEOREM 7.1. *If G is a planar flow graph with variable sources and sinks that lie on at most k faces of G, then a maximum flow for G can be computed sequentially in $O(k^2 n^{1.5} \log^2 n)$ time. In parallel, the running time is $O(k^2 \log^5 n)$ using $O(n^{1.5})$ processors in the CRCW model.*

REFERENCES

[1] C. BERGE AND A. GHOUILA-HOURI, *Programming, Games and Transportation Networks*, John Wiley, New York, 1965.

[2] S. EVEN, *Graph Algorithms*, Computer Science Press, Potomac, Maryland, 1979.

[3] L. R. FORD AND D. R. FULKERSON, *Maximal flow through a network*, Canad. J. Math., 8 (1956), pp. 399–404.

[4] G. N. FREDERICKSON, *Fast algorithms for shortest paths in planar graphs with applications*, SIAM J. Comput., 16 (1987), pp. 1004–1022.

[5] R. HASSIN, *Maximum flows in (s, t) planar networks*, Inform. Process. Lett., 13 (1981), p. 107.

[6] R. HASSIN AND D. B. JOHNSON, *An $O(n \log^2 n)$ algorithm for maximum flow in undirected planar networks*, SIAM J. Comput., 14 (1985), pp. 612–624.

[7] H. GAZIT AND G. L. MILLER, *A parallel algorithm for finding a separator in planar graphs*, Proc. 28th Symposium on Foundations of Computer Science, Los Angeles, CA, 1987, pp. 238–248.

[8] ———, *A deterministic parallel algorithm for finding a separator in planar graphs*, manuscript.

[9] M. GOLDBERG AND T. SPENCER, *Constructing a maximal independent set in parallel*, SIAM J. Discrete Math., 2 (1989), pp. 322–328.

[10] L. GOLDSCHLAGER, R. SHAW, AND J. STAPLES, *The maximum flow problem is log space complete for P*, Theoret. Comput. Sci., 21 (1982), pp. 105–111.

[11] J. E. HOPCROFT AND R. E. TARJAN, *Efficient planarity testing*, J. Assoc. Comput. Mach., 21 (1974), pp. 549–568.

[12] A. ITAI AND Y. SHILOACH, *Maximum flow in planar networks*, SIAM J. Comput., 8 (1979), pp. 135–150.

[13] L. JANIGA AND V. KOUBEK, *A note on finding cuts in directed planar networks by parallel computation*, Inform. Process. Lett., 21 (1985), pp. 75–78.

[14] D. B. JOHNSON, *Parallel algorithms for minimum cuts and maximum flows in planar networks*, J. Assoc. Comput. Mach., 34 (1987), pp. 950–967.

[15] D. B. JOHNSON AND S. VENKATESAN, *Using divide and conquer to find flows in directed planar networks in* $O(n^{1.5} \log n)$ *time*, Proc. 20th Allerton Conference on Communication, Control and Computing, University of Illinois, Urbana-Champaign, Urbana, IL, 1982, pp. 898–905.

[16] R. M. KARP, E. UPFAL, AND A. WIGDERSON, *Constructing a perfect matching is in random NC*, Combinatorica, 6 (1986), pp. 35–48.

[17] P. W. KASTELEYN, *Graph theory and crystal physics*, in Graph Theory and Theoretical Physics, F. Harary, ed., Academic Press, New York, 1967, pp. 43–110.

[18] S. KHULLER AND J. NAOR, *Flow in planar graphs with vertex capacities*, Algorithmica, 11 (1994), pp. 200–225.

[19] P. N. KLEIN AND J. H. REIF, *An efficient parallel algorithm for planarity*, J. Comput. System Sci., 37 (1988), pp. 190–246.

[20] R. J. LIPTON, D. J. ROSE, AND R. E. TARJAN, *Generalized nested dissection*, SIAM J. Numer. Anal., 16 (1979), pp. 346–358.

[21] R. J. LIPTON AND R. E. TARJAN, *Applications of a planar separator theorem*, SIAM J. Comput., 9 (1980), pp. 615–627.

[22] N. MEGIDDO, *Optimal flows in networks with multiple sources and sinks*, Math. Programming, 7 (1974), pp. 97–107.

[23] ———, *A good algorithm for lexicographically optimal flows in multi-terminal networks*, Bull. Amer. Math. Soc., 83 (1977), pp. 407–409.

[24] K. MULMULEY, U. V. VAZIRANI, AND V. V. VAZIRANI, *Matching is as easy as matrix inversion*, Combinatorica, 7 (1987), pp. 105–113.

[25] T. NISHIZEKI AND N. CHIBA, *Planar graphs: Theory and algorithms*, in Ann. Discrete Math., Vol. 32, North-Holland Mathematical Studies, 1988.

[26] V. PAN AND J. H. REIF, *Fast and efficient parallel solution of sparse linear systems*, Tech. report 88–19, Computer Science Department, State University of New York at Albany, Albany, New York, 1988.

[27] ———, *Fast and efficient solution of path algebra problems*, J. Comput. System Sci., 38 (1980), pp. 494–510.

[28] ———, *The parallel computation of minimum cost paths in graphs by stream contraction*, Inform. Process. Lett., 40 (1991), pp. 79–83.

[29] V. RAMACHANDRAN, *Flow value, minimum cuts and maximum flows*, unpublished manuscript.

[30] J. H. REIF, *Minimum* $s-t$ *cut of a planar undirected network in* $O(n \log^2 n)$ *time*, SIAM J. Comput., 12 (1983), pp. 71–81.

[31] Y. SHILOACH AND U. VISHKIN, *An* $O(\log n)$ *parallel connectivity algorithm*, J. Algorithms, 3 (1982), pp. 57–67.

[32] V. V. VAZIRANI, *NC algorithms for computing the number of perfect matchings in* $K_{3,3}$*-free graphs and related problems*, Inform. and Comput., 80 (1989), pp. 152–164.

SIAM J. COMPUT.
Vol. 24, No. 5, pp. 1018–1035, October 1995

005

A SUBEXPONENTIAL ALGORITHM FOR ABSTRACT OPTIMIZATION PROBLEMS*

BERND GÄRTNER†

Abstract. An *abstract optimization problem* (AOP) is a triple $(H, <, \Phi)$ where H is a finite set, $<$ is a total order on 2^H, and Φ is an oracle that, for given $F \subseteq G \subseteq H$, either reports that $F = \min_<\{F' \mid F' \subseteq G\}$ or returns a set $F' \subseteq G$ with $F' < F$. Solving the problem means finding the minimum set in H. We present a randomized algorithm that solves any AOP with an expected number of at most

$$e^{2\sqrt{n}+O(\sqrt[4]{n}\ln n)}$$

oracle calls, $n = |H|$. In contrast, any deterministic algorithm needs to make $2^n - 1$ oracle calls in the worst case.

The algorithm is applied to the problem of finding the distance between two n-vertex (or n-facet) convex polyhedra in d-space, and the computation of the smallest ball containing n points in d-space; for both problems we give the first subexponential bounds in the arithmetic model of computation.

Key words. computational geometry, smallest enclosing ball, distance between convex polyhedra, local optimization, randomized algorithm

AMS subject classifications. 68Q20, 68Q25, 68U05, 90C25, 90C27

1. Introduction.

Three geometric optimization problems. Recently, Sharir and Welzl [22] described an abstract class of problems, so-called *LP-type problems*, that are efficiently solvable by a simple randomized algorithm. The typical LP-type problems are geometric optimization problems, related in spirit to the "master" problem of *linear programming*:

(LP) Given a convex polyhedron $\mathcal{P}$, specified by n halfspaces in d-space, and a d-vector v, find a point $p \in \mathcal{P}$ extreme in direction v.

The geometric formulation is chosen to keep notation consistent with another important LP-type problem, namely, finding the *distance between convex polyhedra*:

(POLYDIST) Given two convex polyhedra $\mathcal{P}$ and $\mathcal{Q}$, specified by n points (or n halfspaces) in d-space, find points $p \in \mathcal{P}$, $q \in \mathcal{Q}$ with $||p - q|| = \text{dist}(\mathcal{P}, \mathcal{Q}) := \min\{||p' - q'|| \mid p' \in \mathcal{P}, q' \in \mathcal{Q}\}$.

The *minimum spanning ball problem* looks somewhat different, but nevertheless fits into the LP-type framework:

(MINIBALL) Given n points in d-space, determine the center and radius of the smallest ball containing all the points.

POLYDIST and MINIBALL can easily be cast in the form of a convex program with only one constraint being nonlinear, and the general method of Grötschel, Lovász, and Schrijver [10] can be applied to give polynomial algorithms for all three problems in the *Turing machine model*. This means that they can be solved in time polynomial in n, d, and the encoding length of the input numbers.

The *arithmetic model* (or random access machine (RAM) model)—widely used in computational geometry—expresses the run time in terms of the overall number of elementary (arithmetic) operations that are performed during the algorithm, where an arithmetic operation

*Received by the editors May 25, 1993; accepted for publication (in revised form) April 20, 1994. A preliminary version of this paper appeared in the Proceedings of the 33rd Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, California, 1992, pp. 464–472.

†Institut für Informatik, Freie Universität Berlin, Takustraße 9, 14195 Berlin, Germany (gaertner@inf.fu-berlin.de). Part of this work was done while the author was a member of the Graduiertenkolleg "Algorithmische Diskrete Mathematik," supported by Deutsche Forschungsgemeinschaft grant We 1265/2-1; this work has also been supported by the ESPRIT Basic Research Action Program 7141 of the EU (ALCOM II).

is charged unit cost, regardless of the sizes of the numbers involved. Thus the run time in the arithmetic model is a function only of the *number* of input numbers and does not depend on their encoding lengths. Usually the arithmetic model is enhanced with the "fairness assumption" that unit cost applies only to operations involving numbers which are not much larger than the input numbers (which here means that the encoding length does not exceed the maximum encoding length over all input numbers by more than a constant multiple). Bounds obtained in this model are called *combinatorial bounds*, and in contrast to the Turing machine model, there are no polynomial combinatorial bounds (also called *strongly polynomial* bounds) known for any of the three problems listed above (for details concerning the computational models see [10, pp. 32–33]).

In this paper we will be concerned with the arithmetic model of computation, and the bounds we develop are combinatorial. Although we do not succeed in proving polynomial bounds, we achieve substantial progress by showing that POLYDIST and MINIBALL have subexponential combinatorial complexity (for LP this had already been established; see [18]); all previous bounds were exponential.

The generic LP-type problem can be solved in a *local optimization* fashion, i.e., if a proposed solution is not yet optimal, one can locally improve on it by solving a "small" subproblem. This basic property underlies the algorithm in [22] that solves the aforementioned problems in time linear in n but exponential in d. This bound is asymptotically optimal if d is constant and still reasonable if d is small compared to n.[1] However, as soon as d gets only moderately large with respect to n, the bound becomes unsatisfactory, and until recently no algorithms were known to have combinatorial complexity less than exponential in d *and* n for any of the three problems. Then Kalai [11] and Matoušek, Sharir, and Welzl [18] independently came up with subexponential bounds for LP. The bound in [18] was obtained by tuning the analysis of the algorithm in [22], and although this algorithm can solve any LP-type problem, the subexponential analysis is valid only for LP. This is a result of the fact that, at least under certain standard assumptions, the "small" instances of LP ($n = d + 1$) can easily be solved in polynomial time, while for POLYDIST and MINIBALL the small problems are not substantially easier than the general ones.

In this paper we will show that the subexponential bound for LP is actually induced only by the local optimization property and does not rely on additional convenient features of the small instances. Consequently, POLYDIST and MINIBALL can be solved as efficiently as LP by our method. The tool is the framework of the *abstract optimization problems* (AOPs) that captures the spirit of local optimization in a generic setting.

Basically, an AOP consists of a finite set H with a total order on 2^H and an oracle that answers the following query: for given $F \subseteq G \subseteq H$, does there exist a set $F' \subseteq G$ with $F' < F$? If the answer is yes, such a set is returned as a witness. Solving the problem means finding the minimum set in the total order, and we want to bound the number of oracle queries needed to do this for any given AOP.

The algorithm we develop is *randomized*, i.e., we count the number of oracle queries averaged over internal coin flips performed by the algorithm. We do not average over an input distribution; the expectation we get is valid for any input (in particular, there is no input that forces the algorithm to perform poorly). Moreover, randomization is crucial: if information about the linear order $<$ can be obtained from the oracle only, no deterministic algorithm can beat the trivial bound of $2^{|H|} - 1$ oracle queries in the worst case. Although this does not mean much for a specific instance of an AOP, it gives evidence that randomized algorithms may be potentially more powerful than deterministic ones in this situation.

[1] Such assumptions are frequently made in computational geometry, and some bounds cited here make full sense only if d is substantially smaller than n.

The paper is organized as follows: In the rest of the introduction we give a brief survey on results concerning the combinatorial complexity of the LP, POLYDIST, and MINIBALL problems; in §2 we discuss Sharir and Welzl's LP-type problems and point out why the subexponential bound for LP established in [18] does not hold for POLYDIST and MINIBALL. In §3 we formally introduce our abstract framework and state the main result of the paper that implies the subexponential bounds for POLYDIST and MINIBALL. Section 4 contributes the more technical part; it proves the deterministic lower bound and the randomized upper bound by presenting an algorithm in the abstract framework. Section 5 provides a concluding discussion.

Linear programming. LP problems are probably the best-understood optimization problems. There exists a vast amount of literature; the reader is referred to [20] for an introduction. There are several methods for solving LP-problems that are efficient in practice, the most popular one being the *simplex algorithm* that was introduced by Dantzig [6] in 1951. Its good performance on practical problems is in contrast to a result by Klee and Minty [15], who showed that a frequently used pivot rule leads to exponential behavior in d in the worst case (modifying the Klee–Minty construction, this was extended to other pivot rules; see Klee and Kleinschmidt [14]). The simplex algorithm is a "combinatorial" algorithm in the sense that its behavior depends on the ordering of the vertices of the polyhedron along the optimization direction but not on their specific coordinates.

In contrast to this, the algorithms of Khachiyan (ellipsoid method [13]) and Karmarkar (interior point method [12]) are approximation algorithms which arrive at the desired optimum in time depending on the input precision. Both algorithms give weakly polynomial time bounds (i.e., polynomial in the Turing machine model), thus showing that linear programming belongs to the complexity class P. This caused considerable excitement in 1979 when Khachiyan's result became known. The quest for a strongly polynomial algorithm continues, but it is not generally believed that polynomial combinatorial bounds can be achieved.

Nevertheless, there has been substantial progress on the combinatorial complexity of linear programming over the last decade, mainly coming from computational geometry. Megiddo [17] was the first to show that LP can be solved in time linear in the number n of constraints (with a doubly exponential dependence on d). Subsequently, the dependence on d has been improved step-by-step while keeping the bound linear in n; see Dyer [7], Clarkson [1], Seidel [21], and Sharir and Welzl [22]. Currently, the best (randomized) algorithm combines the recent subexponential results in [11] and [18] with an algorithm by Clarkson [2]. This gives a bound of

$$O(d^2 n + e^{O(\sqrt{d \log d})}).$$

The best deterministic algorithm is by Chazelle and Matoušek [4] and is obtained by "derandomizing" Clarkson's algorithm. Its run time is $O(d^{O(d)} n)$.

Distance between convex polyhedra. This problem—and the important special case that one polyhedron is a single point—is an instance of a quadratic optimization problem, and it has applications, e.g., in motion planning (collision testing); there are heuristics for it without time analysis (see, e.g., Wolfe [25] for the special case and Sekitani and Yamamoto [23] for the general case). A first nontrivial randomized time bound of $O(n^{\lfloor d/2 \rfloor})$ was given by Clarkson [3] (provided the polyhedra are specified by n points). Applying the algorithm in [22], this can be improved to give the best bound so far of $O(d^3 2^d n)$, which is linear in n but still exponential in d. Our algorithm will establish the same subexponential bound as stated above for linear programming.

Minimum spanning ball. It was observed early that MINIBALL (which is a prototype problem in facility location) has a structure similar to LP and thus can also be solved in time

linear in the number of points by the techniques in [17] and [7] (with the same dependence on d as in the case of LP). As observed by Welzl [24], the LP algorithm of Seidel [21] also applies to MINIBALL and the same holds for the algorithms of Clarkson and Sharir and Welzl. As in the case of POLYDIST, subexponential run time could not yet be shown but will be established in this paper.

2. Basics and terminology.

LP-type problems. To begin with, let us briefly review the concept of the *LP-type problems* introduced in [22], where the reader can also find how LP itself fits into the framework. Consider the MINIBALL problem first and let H be a set of n points in d-space. For $G \subseteq H$ denote by $w(G)$ the radius of the smallest ball containing the points in G. It is well known that this ball is unique and that for $e \in H$, $w(G) < w(G \cup \{e\})$ if and only if e lies outside the ball determined by G. From this it is easily seen that the following two properties hold for all $F \subseteq G \subseteq H$:

(i) $w(F) \leq w(G)$;

(ii) if $w(F) = w(G)$, then $w(F) < w(F \cup \{e\}) \Leftrightarrow w(G) < w(G \cup \{e\})$ for all $e \in H$.

In general, any pair (H, w) satisfying (i) and (ii) is called an LP-type problem. A *basis* is a set $B \subseteq H$ with $w(B') < w(B)$ for all $B' \subsetneq B$. A basis of $G \subseteq H$ is a basis $B \subseteq G$ such that $w(B) = w(G)$. The *combinatorial dimension* of (H, w) is the maximum cardinality of any basis. In this framework, MINIBALL has combinatorial dimension at most $d+1$, because any minimum ball spanned by a set $G \subseteq H$ is already determined by at most $d+1$ points of G on the boundary.

POLYDIST gives rise to an LP-type problem as follows. (We assume for the rest of the paper that polyhedra $\mathcal{P}$ and $\mathcal{Q}$ are given by point sets P and Q, $P \cap Q = \emptyset$, $|P \cup Q| = n$. So $\mathcal{P}$ and $\mathcal{Q}$ are actually polytopes with $\mathcal{P} = \text{conv}(P)$, $\mathcal{Q} = \text{conv}(Q)$; the case where $\mathcal{P}$ and $\mathcal{Q}$ are specified by halfspaces is similar but requires more technicalities.)

For $P' \cup Q' \subseteq P \cup Q$, let

$$w(P' \cup Q') := \text{dist}(\text{conv}(P'), \text{conv}(Q')).$$

For P' or Q' empty, w is set to ∞. Now properties (i) and (ii) (for $w(F) = w(G) < \infty$) hold for the pair $(P \cup Q, w)$, but with $<$ and $\leq$ replaced by $>$ and $\geq$, respectively. Property (i) is obviously satisfied, so let us prove property (ii). Assume $w(P' \cup Q') = \rho$, $0 < \rho < \infty$. Then there exists a unique pair $h_{P'}, h_{Q'}$ of parallel supporting hyperplanes of distance ρ which are perpendicular to any vector $p - q$ with $p \in \text{conv}(P')$, $q \in \text{conv}(Q')$, and $||p - q|| = \rho$.

Denote by $h^+_{P'}$ and $h^+_{Q'}$ the closed halfspaces containing P' and Q', respectively. For a point $e \in P - P'$ ($Q - Q'$) we have $w(P' \cup Q' \cup \{e\}) < w(P' \cup Q')$ if and only if e does not lie in $h^+_{P'}$ ($h^+_{Q'}$) (Fig. 1). This implies (ii).

It is a straightforward exercise to show that the following lemma holds.

LEMMA 2.1. *In the LP-type framework, the combinatorial dimension of* POLYDIST *defined by polytopes $\mathcal{P}$ and $\mathcal{Q}$ in d-space satisfies $\delta \leq d + 2$ (if $\mathcal{P}$ and $\mathcal{Q}$ are disjoint, $\delta \leq d + 1$).*

The following theorem is the main result of [18].[2]

THEOREM 2.2. *Let (H, w) with $|H| = n$ be an LP-type problem of combinatorial dimension δ, and denote by t_{small} the time necessary to compute a basis of G and its value $w(G)$ for $|G| = \delta + 1$. Then a basis of H and $w(H)$ can be computed in time*

$$O(t_{\text{small}}\, n e^{2\sqrt{\delta \ln n}}).$$

[2]The bounds are actually slightly better (and more complicated) than what we cite here.

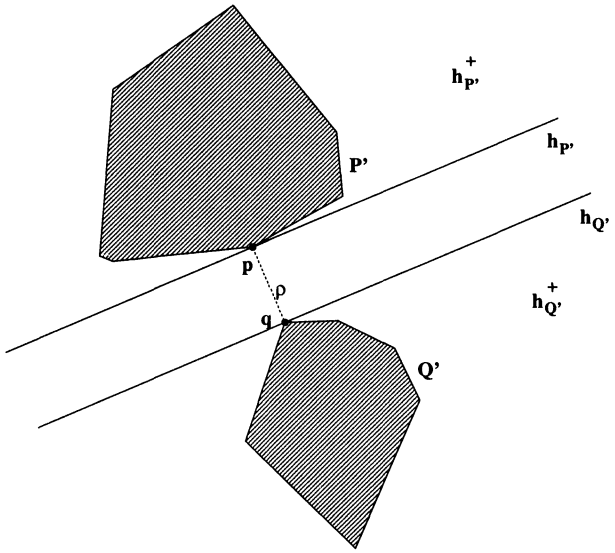


FIG. 1. *The* POLYDIST *problem.*

As the theorem shows, the time bound for an LP-type problem crucially depends on the complexity of the "small" instances, and it is not clear that solving them is a substantially easier task than solving the whole problem. For d-dimensional LP, the small problem basically consists of solving a linear program with $d + 1$ constraints. Assuming that the program is bounded and in general position, its solution is determined as the intersection of exactly d of the constraint hyperplanes, which implies a polynomial procedure (for details see [18]).

In case of POLYDIST and MINIBALL, the small problems are as follows:

(SMALL_POLYDIST) Given two convex polyhedra $\mathcal{P}$ and $\mathcal{Q}$, specified by at most $d + 3$ points $P \cup Q$ in d-space, determine their distance and a basis, i.e., a minimal subset $P' \cup Q'$ determining the same distance.

(SMALL_MINIBALL) Given a set H of at most $d + 2$ points in d-space, determine the radius of the smallest ball containing H and a basis, i.e., a minimal subset B determining the same ball.

In contrast to LP, it is no longer true that the cardinality of a basis is known a priori, even under nondegeneracy assumptions. In case of POLYDIST, a basis of $P \cup Q$ may consist of any number of points between 2 and $d + 2$. Consequently, straightforward checking of every candidate basis as in the case of LP may require the examination of $\Theta(2^d)$ subsets just to solve SMALL_POLYDIST, which gives nothing better than an exponential algorithm for the whole problem. The same difficulty arises in the MINIBALL problem: a minimum spanning ball may be determined by any number of points between 2 and $d + 1$ on its boundary, so again SMALL_MINIBALL is not efficiently solvable by the trivial method. By embedding SMALL_POLYDIST into the AOP framework we will be able to solve it in time $e^{O(\sqrt{d})}$, and the same complexity will be achieved for SMALL_MINIBALL, which will turn out to be a special case of SMALL_POLYDIST. Plugging this into Theorem 2.2 will also give subexponential bounds for the corresponding "large" problems.

3. Abstract optimization problems. Let us repeat the definition of an AOP in a formal way to have the accurate terminology available.

DEFINITION 3.1. *An AOP is a triple* $(H, <, \Phi)$, *where* H *is a finite set,* $<$ *is a total order on* 2^H, *and* Φ *is a mapping*

$$\Phi : \mathcal{H} \to H,\ \mathcal{H} := \{(F, G) \mid F \subseteq G \subseteq H\}$$

with the following property:

$$\Phi(F, G) = F \text{ if and only if } F = \min_{<}\{F' \mid F' \subseteq G\},$$

$$G \supseteq \Phi(F, G) < F \text{ otherwise.}$$

For $G \subseteq H$ *let* $\mathrm{opt}(G)$ *denote* $\min_{<}\{F \mid F \subseteq G\}$. Solving *the AOP means finding* $\mathrm{opt}(H)$.

We achieve the following results (proofs are postponed to the technical section).

THEOREM 3.2 (deterministic lower bound). *For any* deterministic *algorithm* $\mathcal{A}$ *that solves all AOPs on a set* H, $|H| = n$, *there exists an AOP* $(H, <, \Phi)$ *that cannot be solved by* $\mathcal{A}$ *with less than* $2^n - 1$ *oracle queries.*

THEOREM 3.3 (randomized upper bound). *There exists a* randomized *algorithm that solves any AOP on a set* H, $|H| = n$, *with an expected number of at most* $e^{2\sqrt{n}+O(\sqrt[4]{n}\ln n)}$ *oracle queries.*

Using the terminology from the previous section, we will now demonstrate how POLYDIST, defined by point sets P and Q, fits into the AOP framework. The ground set H is $P \cup Q$ and the total order $<$ on 2^H is defined as follows: the bases (in the LP-type sense) are ordered by their w-values with ties broken arbitrarily and any nonbasis is larger than any basis. This definition ensures that $\mathrm{opt}(H)$ is indeed a basis of $P \cup Q$ and the nonbases are not of interest.

It remains to describe the oracle Φ. Φ will only be called on pairs (F, G) where F is a basis and will deliver only bases. We need more terminology: For a point p and $P' \subseteq P$ with $p \in \mathrm{conv}(P')$ (equivalently for q, Q') let $f(p, P') \subseteq P'$ be inclusion-minimal with $p \in \mathrm{conv}(f(p, P'))$. For $F' = P' \cup Q' \subseteq P \cup Q$ let (p_F, q_F) be a pair of points realizing the distance between the affine hulls of P' and Q', i.e., $||p_{F'} - q_{F'}|| = \mathrm{dist}(\mathrm{aff}(P'), \mathrm{aff}(Q'))$ (Fig. 2 (a)). This point pair need not be unique in general; if $P' \cup Q'$ is a basis, however, it is.

Now we can implement the oracle $\Phi(F, G)$, $F = P' \cup Q' \subseteq P \cup Q = G$. Its idea is to start with points p and q realizing $w(F)$; provided that one can improve over F at all (which is the case if and only if a point of P lies in the complement of $h^+_{P'}$ or a point of Q lies in the complement of $h^+_{Q'}$—we say that F is *violated* by that point), a loop is performed in which p and q move along straight lines, thereby decreasing $||p - q||$, until a stable position, i.e., a new basis, is obtained. In the generic step of the loop there are points p, q and sets P', Q' such that $p \in \mathrm{conv}(P'), q \in \mathrm{conv}(Q')$. By definition $||p - q|| \geq ||p_{F'} - q_{F'}||$, where $F' = P' \cup Q'$, so by moving p and q simultaneously along straight lines towards $p_{F'}$ and $q_{F'}$, respectively, their distance decreases in a monotone fashion. The movement stops if either the destination points are reached (in which case F' is a new basis) or one of p and q hits the boundary of $\mathrm{conv}(P')$ or $\mathrm{conv}(Q')$, respectively; in this case, the loop continues after setting P' to $f(p, P')$ and Q' to $f(q, Q')$, which decreases $|P' \cup Q'|$ by at least one (Fig. 2 (b)). The pseudocode[3] for the procedure just described is as follows.

[3]The variant we use here is adapted from the book *Introduction to Algorithms* by T. H. Cormen, C. E. Leiserson, and R. L. Rivest, MIT Press, Cambridge, MA, 1990.

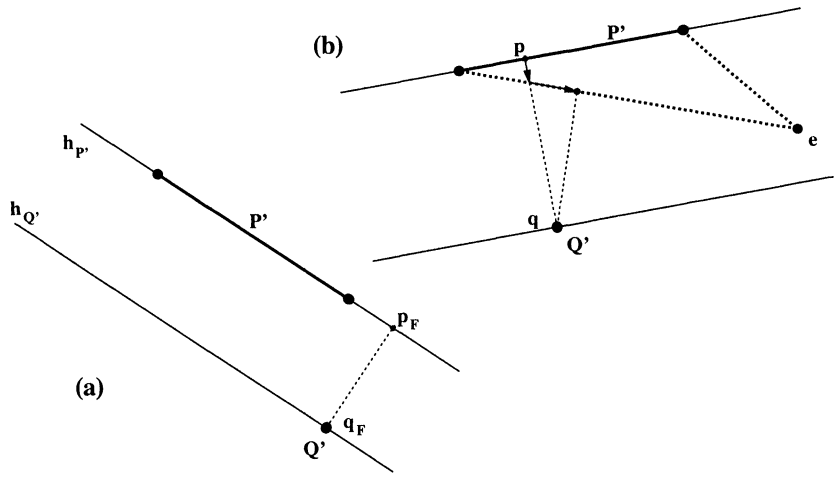


FIG. 2. *The* POLYDIST *oracle.*

```
Φ(F, G)                                   ▷F basis, F = P' ∪ Q' ⊆ P ∪ Q = G
 1   (p, q) := (p_F, q_F)                 ▷p_F, q_F ∈ conv(P'), conv(Q'), unique
 2   for all e ∈ G − F
 3       do if e violates F
 4           then case e of
 5                        ∈ P : P' ← P' ∪ {e}
 6                        ∈ Q : Q' ← Q' ∪ {e}
 7                    loop F' ← P' ∪ Q'
 8                        (p_λ, q_λ) := (p, q) + λ((p_F', q_F') − (p, q))
 9                        μ ← max{λ | p_λ ∈ conv(P'), q_λ ∈ conv(Q')}
10                        if μ > 1
11                           then return F'
12                        (p, q) ← (p_μ, q_μ)
13                        (P', Q') ← (f(p, P'), f(q, Q'))
14   return F
```

From the discussion above, termination of the procedure follows. To see the correctness, observe that the pair $(p_{F'}, q_{F'})$ in line 8 is unique in any iteration of the loop, since $\text{aff}(P')$ and $\text{aff}(Q')$ are neither parallel (unless $|P'| = 1$ or $|Q'| = 1$, in which case uniqueness holds anyway) nor have a nontrivial intersection. From this it follows that the set F' finally returned in line 11 is indeed a basis.

$||p - q||$ strictly decreases in every iteration of the loop, so the basis finally returned has smaller w-value than F.

We are interested in the run time of this oracle when called on a "small" problem, i.e., $|G| \leq d + 3$. The following bound is certainly not best possible; the interesting fact is that it is polynomial.

LEMMA 3.4. *Let G be a set of at most $d + 3$ points in d-space. The oracle $\Phi(F, G)$ can be implemented to run in time $O(d^5)$.*

Proof. The first phase (checking whether an improvement over F is possible) amounts to the computation of one scalar product per point $e \in G - F$ and, therefore, can be done in time $O(d^2)$. The loop is executed $O(d)$ times; in each iteration, points $p_{F'}$, $q_{F'}$ can be

computed in time $O(d^3)$ by solving a system of linear equalities. In order to find μ, we have to intersect two lines with the bounding hyperplanes of $\text{conv}(P')$ and $\text{conv}(Q')$, respectively. From the fact that F was a basis it follows that these polytopes are simplices, so there are no more than $d+1$ bounding hyperplanes per polytope and each intersection can be computed in time $O(d^3)$ again. This process also gives $f(p, P')$ and $f(q, Q')$ as a by-product. □

By arbitrarily choosing two vertices in the beginning, one from each polytope, we get an initial basis for the oracle. By Theorem 3.3 a SMALL_POLYDIST problem can be solved in time $e^{O(\sqrt{d})}$, and together with Theorem 2.2 this gives the following theorem.

THEOREM 3.5. *The distance between two n-vertex polytopes in d-space can be computed in time* $O(ne^{(2+o(1))(\sqrt{d \ln n})})$.

We remark that the same result holds for two n-facet convex polyhedra. Note that the contribution from the small problem is hidden in the $o(1)$ term of the exponent.

We get the same bound for MINIBALL.

THEOREM 3.6. *The smallest enclosing ball of a set of n points in d-space can be computed in time* $O(ne^{(2+o(1))(\sqrt{d \ln n})})$.

For this it suffices to show that a SMALL_MINIBALL problem can be solved as efficiently as SMALL_POLYDIST and, as it turns out, both problems are strongly related; the following correspondence can be found, e.g., in [19].

THEOREM 3.7. *Let P be a set of $d+1$ affinely independent points in d-space with circumcenter q_0. The center q_1 of the smallest ball containing P is the point in* $\text{conv}(P)$ *with minimum distance to q_0.*

Thus, in order to solve SMALL_MINIBALL on $d+2$ points, compute—by solving $d+2$ SMALL_POLYDIST problems with one polytope as a single point—all the smallest balls spanned by $d+1$ of the points, and compare their radii. In case the input consists of d points or less, the problem restricts to a lower-dimensional one inside the affine hull of the points. Note that the circumcenter of $d+1$ points can be computed in time $O(d^3)$.

4. Bounds for abstract optimization problems. This section contains the proofs of Theorems 3.2 and 3.3. The deterministic lower bound follows from an adversary argument. For the randomized upper bound we present an algorithm along with a careful analysis. Let us start with the lower bound.

4.1. The deterministic lower bound. Let H be an n-element set and suppose we have a deterministic algorithm for solving any AOP $(H, <, \Phi)$.

We start the algorithm on a problem $(H, <_0, \Phi_0)$ with $<_0$ and Φ_0 not yet determined, and we argue that an adversary answering the oracle queries can construct $<_0$ and Φ_0 "online" in such a way that the algorithm is forced to step through at least $2^n - 1$ queries. When supplied with a query pair (F, G), the adversary will output an answer $F' = \Phi_0(F, G)$ according to two simple rules:

(i) the answer F' is consistent with the previous ones, i.e., there exists an AOP such that the current and all previous queries have been answered correctly with respect to this AOP.

(ii) $F' = F$ if and only if there is no other consistent answer.

It is easy to see that the adversary always has a consistent answer, so the algorithm steps through a sequence of queries with pairs (F, G) and finally stops. Suppose that less than $2^n - 1$ queries have been performed. Then there are two sets F_1 and F_2 which have never been the first component of a query pair. We will show that it is consistent with all answers to assume that $F_1 = \text{opt}(H)$. The same holds for F_2, so whatever the algorithm outputs, there is an AOP that is not correctly solved. Hence the adversary can force the algorithm to step

through at least $2^n - 1$ queries, which means that it performs that many queries on the AOP that has implicitly been constructed by the adversary.

We are left to prove that $F_1 = \text{opt}(H)$ is consistent. Clearly, this choice can fail only if some answer has revealed the existence of a smaller set. Since there was no query pair (F_1, G), the only possibility remaining for this to happen is that some query (F, G) with $F_1 \subseteq G$ has been answered by F, thus establishing $F < F_1$. But in the first query of this type F_1 was not bounded from below yet, so it could have been returned instead of F, a contradiction of rule (ii).

4.2. The randomized upper bound. We present a randomized algorithm that solves any given AOP $(H, <, \Phi)$ on an n-element set H with an expected number of at most $e^{2\sqrt{n}+O(\sqrt[4]{n}\ln n)}$ calls to the oracle Φ. Up to an $O(n)$ overhead caused by set operations, the actual run time of the algorithm will be dominated asymptotically by the time spent on the oracle calls, so this is a reasonable measure of complexity. The algorithm will eventually output $\text{opt}(H)$, but the generic step in the recursive procedure is the computation of $\text{opt}(G)$ for $G \subseteq H$. Together with G, a set $F \subseteq G$ is maintained; throughout the section we will refer to F as an *estimate* for the solution that will be improved over and over again until it coincides with the desired minimum. Our algorithm combines ideas of both Kalai's and Matoušek, Sharir, and Welzl's subexponential LP algorithms, and substantially generalizes their applicability.

The section proceeds in stages: in the first stage we introduce a trivial algorithm and the basic terminology; the second stage presents an algorithm that, although it works only modulo a hypothetic subroutine, features the heart of the final algorithm and its subexponential analysis; stage three describes a "working" algorithm that will be obtained by "approximating" the subroutine to a reasonable extent.

Getting started. To acquaint ourselves with the problem, we give the obvious deterministic method for obtaining $\text{opt}(G)$ in the presence of an estimate F.

```
AOP_DET(F, G)
1   repeat F' ← F
2          F ← Φ(F, G)
3   until F = F'
4   return F'
```

To solve the problem on G, AOP_DET has to call the oracle $2^{|G|} - 1$ times in the worst case, and as we have seen in the previous section, it shares this exponential behavior with any algorithm that is deterministic or calls the oracle only on pairs of the form $(*, G)$. Consequently, the method we describe now is randomized and uses oracle queries of the form $\Phi(*, G')$ for certain subsets $G' \subseteq G$. It will have one very intuitive property in common with AOP_DET: the better the estimate F, the faster the algorithm for G. A natural measure for the quality of F in this context is the *rank* of F with respect to G, defined by

$$\text{rank}(F, G) := \#\{F' \subseteq G \mid F' < F\}.$$

A somewhat coarser but related measure is the *dimension* of a pair (F, G).

DEFINITION 4.1. *For $F \subseteq G$, an element $e \in G$ is* enforced *in (F, G) if $F < \text{opt}(G - \{e\})$ (and this implies $e \in F$). Otherwise it is* free. *The* domain *of (F, G) is the set of free elements, i.e.,*

$$\mathcal{D}(F, G) := \{e \in G \mid \exists F' \subseteq G - \{e\}, F' \leq F\}.$$

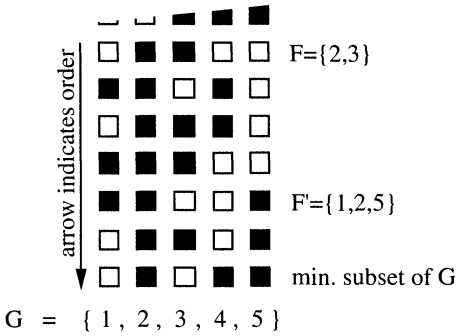


FIG. 3. *Part of an AOP on* 5 *elements.*

For a free element e, an estimate $F' \subseteq G - \{e\}$ *with* $F' \leq F$ *is called a* witness *for e. Finally, the* dimension *of* (F, G) *is the size of its domain,*

$$\dim(F, G) := |\mathcal{D}(F, G)|.$$

The enforced elements in (F, G) are exactly the ones contained in *every* estimate $F' \subseteq G$ with $F' \leq F$, while the free elements have at least one witness $F' \subseteq G - \{e\}$ with $F' \leq F$.

As an example consider Fig. 3. Subsets of G are visualized by elements of $\{\blacksquare, \square\}^{|G|}$. (F, G) enforces element 2 while (F', G) enforces 2 and 5. Consequently, $\mathcal{D}(F, G) = \{1, 3, 4, 5\}$, $\mathcal{D}(F', G) = \{1, 3, 4\}$, and $\dim(F, G) = 4, \dim(F', G) = 3$. Note that we always have $\dim(F, G) \geq \log_2(\text{rank}(F, G) + 1)$.

The following monotonicity lemma is an immediate consequence of the definitions and (although quite obvious) forms the background of the analysis in the next stage.

LEMMA 4.2. (i) *If* $F' \leq F$ *then* $\mathcal{D}(F', G) \subseteq \mathcal{D}(F, G)$.

(ii) *If* $F \subseteq G \subseteq G'$ *then* $\mathcal{D}(F, G) \subseteq \mathcal{D}(F, G')$.

The basic algorithm. Now we are able to describe a first basic version of our algorithm. We feel that the main idea behind the subexponential analysis can be explained most clearly by assuming that the following subroutine is available.

```
SAMPLE_DOMAIN(F, G)
1   if D(F, G) = ∅
2      then error "empty"
3   choose a random e from D(F, G)
4   choose a witness F_e ≤ F with F_e ⊆ G − {e}
5   return (e, F_e)
```

So SAMPLE_DOMAIN chooses an element which is free in (F, G) at random and outputs it, along with a corresponding witness. Note that the elements in $G - F$ are free with witness F; however, there may be other free elements which are not immediately accessible since they are hidden in F, and it is not clear whether one can find them efficiently. Nevertheless, let us assume for this stage that SAMPLE_DOMAIN comes free.

Using this subroutine, we can formulate a procedure AOP_SD(F, G) to find opt(G) in the presence of estimate F.

```
AOP_SD(F, G)
1   (e, F_e) := SAMPLE_DOMAIN(F, G)
2   if error "empty"
3     then return F
4   F ← AOP_SD(F_e, G − {e})
5   F' ← Φ(F, G)
6   if F = F'
7     then return F'
8   return AOP_SD(F', G)
```

The analysis. Termination and correctness easily follow by observing that the first recursive call solves a subproblem on a smaller set, and in the second one we have $\text{rank}(F', G) < \text{rank}(F, G)$. For fixed H let $T(k)$ be the worst-case expected number of oracle queries performed in a call to AOP_SD(F, G) with $\dim(F, G) = k$ and $G \subseteq H$. We get $T(0) = 0$ and for $k > 0$ we get the following bound.

THEOREM 4.3.

$$T(k) \leq T(k-1) + 1 + \frac{T(k-1) + \cdots + T(0)}{k}.$$

Proof. First of all, it is not hard to see that T is monotone in k. Now we can argue as follows: if (F, G) has dimension k then $\dim(F_e, G - \{e\}) \leq k - 1$, since

$$\mathcal{D}(F_e, G - \{e\}) \cup \{e\} \subseteq \mathcal{D}(F_e, G) \subseteq \mathcal{D}(F, G).$$

This implies that the expected number of oracle queries necessary in the first recursive call is bounded by $T(k-1)$. Another query is done in line 5; the last term finally gives a bound for the second recursive call. To see this, consider $\mathcal{D}(F, G) = \{e_1, \ldots, e_k\}$, ordered in such a way that

$$\text{opt}(G - \{e_1\}) \geq \text{opt}(G - \{e_2\}) \geq \cdots \geq \text{opt}(G - \{e_k\}).$$

If e is chosen to be e_i by SAMPLE_DOMAIN, we get

$$F' < \text{opt}(G - \{e_j\}) \text{ for all } j \leq i,$$

so by Definition 4.1, $e_1, \ldots, e_i$ will be enforced in (F', G). This means

$$\mathcal{D}(F', G) \subseteq \mathcal{D}(F, G) - \{e_1, \ldots, e_i\},$$

so $\dim(F', G) \leq k - i$ and an expected number of no more than $T(k-i)$ oracle queries is performed by AOP_SD(F', G). Since i is equally likely to be any number between 1 and k, we obtain the desired average. □

The heart of the argument is the fact that by choosing a *random* element for the recursion in line 4 of AOP_SD, the dimension halves on the average, while choosing e deterministically can only guarantee a progress of one in the worst case. This basic observation also underlies the subexponential LP-algorithms of Kalai [11] and Matoušek, Sharir, and Welzl [18].

To find an explicit bound for $T(k)$, we majorize $T(k)$ by $t(k) - 1$, where $t(k)$ satisfies

$$t(k) = t(k-1) + \frac{1}{k} \sum_{i=0}^{k-1} t(i)$$

with $t(0) = 1$.

LEMMA 4.4.

$$t(k) = \sum_{i=0}^{k} \binom{k}{i} \frac{1}{i!}.$$

Proof. An easy way to see this is via a nice combinatorial interpretation of $t(k)$ (suggested by P. Flajolet): Consider a permutation $\pi \in S_k$. A subset $R \subseteq \{1, \ldots, k\}$ is called an *increasing chain* in π iff $\forall\, x, y \in R : x < y$ implies $\pi(x) < \pi(y)$. Denote by $s(\pi)$ the number of increasing chains in π. Then $t(k) = E[s]$, the expected number of increasing chains in a random permutation $\pi \in S_k$. The proof of this fact is by induction. For $k = 0$ we have one increasing chain, namely, the empty set. Assume $k > 0$; by the inductive hypothesis the expected number of increasing chains not containing k is $t(k-1)$. The ones containing k are in one-to-one correspondence with the increasing chains in $\{1, \ldots, \pi^{-1}(k) - 1\}$, whose expected number is $t(\pi^{-1}(k) - 1)$. Since $\pi^{-1}(k)$ is equal to i with probability $1/k$, for any $i \in \{1, \ldots, k\}$, we recover the original recurrence for $t(k)$. On the other hand,

$$\begin{aligned} t(k) = E[s] &= \sum_{R} \text{prob}(R \text{ is an increasing chain}) \\ &= \sum_{i=0}^{k} \sum_{|R|=i} \text{prob}(R \text{ is an increasing chain}) = \sum_{i=0}^{k} \binom{k}{i} \frac{1}{i!}, \end{aligned}$$

and the lemma follows. □

COROLLARY 4.5.

$$t(k) \leq e^{2\sqrt{k}}.$$

Proof. Using the inequality $\binom{k}{i} \leq k^i / i!$ we obtain

$$t(k) = \sum_{i=0}^{k} \binom{k}{i} \frac{1}{i!} \leq \sum_{i=0}^{k} \frac{k^i}{i!^2} = \sum_{i=0}^{k} \left(\frac{\sqrt{k}^i}{i!} \right)^2 \leq \left(\sum_{i=0}^{k} \frac{\sqrt{k}^i}{i!} \right)^2 \leq \left(\sum_{i=0}^{\infty} \frac{\sqrt{k}^i}{i!} \right)^2 = e^{2\sqrt{k}}. \quad \square$$

The bound is almost tight; we will come back to this later. For the performance of AOP_SD we get the following theorem.

THEOREM 4.6. AOP_SD *solves any AOP* $(H, <, \Phi)$ *with an expected number of at most* $e^{2\sqrt{|H|}} - 1$ *oracle queries.*

How to sample from the domain. To turn the procedure AOP_SD from the previous paragraph into a working algorithm, we have to do something about the subroutine SAMPLE_DOMAIN. As we have already indicated, the way to deal with it will be to find a reasonably cheap way to "approximate" it. The idea is simple: rather than sampling from the whole domain of a given pair (F, G), we will identify a subset D of the domain, along with the corresponding witnesses, and sample from D only. After plugging in this version of SAMPLE_DOMAIN, the expected performance of AOP_SD will drop off depending on the size of D, which we assume to be a function $r(k)$ of $k = \dim(F, G)$; the recurrence of Theorem 4.3 then becomes

$$T(k) \leq T(k-1) + \frac{T(k-1) + \cdots + T(k - r(k))}{r(k)}.$$

This bound is exponential for $r(k) = O(1)$ but becomes better the closer $r(k)$ is to k. On the other hand, the larger $r(k)$ is, the harder the task of finding the set D is. A reasonable balance is achieved by choosing $r(k)$ proportional to k, say $r(k) = \lceil ck \rceil$, for some fixed $0 < c < 1$

(the exact value will be determined later and will depend on $n = |H|$). We will see that the additional effort for finding $D \subseteq \mathcal{D}(F, G)$ of this size is small, and the algorithm basically preserves its expected performance as previously analyzed.

To construct D when called on a pair (F, G), the algorithm proceeds incrementally; since we know that at least the elements in $G - F$ are free with witness F, we can start off by setting D to $G - F$. In case D is already large enough, we just sample from D and proceed as before. Otherwise we will have to enlarge D by at least one more free element hidden in F; to this end we will step through a sequence of improving oracle queries (in a way to be described later) until a witness for a yet unknown free element is found (or we already end up in $\mathrm{opt}(G)$).

Suppose that in the generic step certain elements in G have already been identified as free in (F, G) and we need to find another free one. The way to do it is as follows: call the algorithm recursively with (F', G), where F' is the current estimate, but supply an additional parameter E that contains all the elements of G whose status is yet unknown (note that $E \subseteq F'$). This recursive call now has two ways to terminate: either it finds the desired solution $\mathrm{opt}(G)$ or, while improving its estimate, discovers an $F'' \subseteq G$ which fails to contain E. This, however, means that the elements in $E - F''$ have been uncovered as free elements with witness F'', so the call has accomplished its task. The key observation is that as long as D is small, the set E of elements with unknown status will be large, and since the recursive call with parameter E terminates as soon as the first estimate F'' that is no longer wedged between E and G appears, it actually operates only on $G - E$ instead of G, which makes it substantially cheaper (this method is a generalization of the idea behind the pivoting strategy Kalai uses in his LP algorithm).

A working algorithm. The generic call will have three parameters $E \subseteq F \subseteq G$, where in the beginning $E = \emptyset$. Let us formulate the procedure AOP(E, F, G) that will either return $\mathrm{opt}(G)$ or deliver an estimate $F' < F$ with $E \not\subseteq F'$ to a higher level in the recursion. The set D is implicitly maintained and comments will refer to it.

```
AOP(E, F, G)                                  ▷ returns opt(G) or F' < F, E ⊈ F'
 1  if E = G                                  ▷ E = F = G ?
 2     then return Φ(F, G)                    ▷ Φ(F, G) = F or E ⊈ Φ(F, G)
 3  E' ← F                                    ▷ elements of unknown status
 4  for all e ∈ G − E'                        ▷ D := G − E', initial free elements
 5     do F_e ← F                             ▷ set witness
 6  while |G − E'| < ⌈c|G − E|⌉               ▷ D still to small, try to enlarge it
 7     do F ←AOP(E', F, G)
 8        if E ⊈ F                           ▷ return to higher level in recursion
 9           then return F
10        if E' ⊈ F                          ▷ new free element(s) found
11           then for all e ∈ E' − F
12                   do F_e ← F               ▷ set witness
13                E' ← E' ∩ F                ▷ update E' (D := D ∪ (E' − F))
14           else return F                   ▷ line 7 has already computed opt(G)
15  choose a random e ∈ G − E'                ▷ sample from D
16  F ←AOP(E, F_e, G − {e})
17  if E ⊈ F                                  ▷ return to higher level in recursion
18     then return F
19  F' ← Φ(F, G)                              ▷ once we get here, F = opt(G − {e})
20  if E ⊈ F' or F = F'                       ▷ check both termination criteria
21     then return F'
22  return AOP(E ∪ {e}, F', G)                ▷ repeat with better estimate
```

Termination of AOP follows by observing that the recursive calls solve smaller problems (measured in terms of $|G - E|$). The correctness of the procedure is not obvious at first sight, but it suffices to inductively check the invariant that E is contained in every estimate up to the terminating one. This is not a priori clear in the recursive call of line 22, where E is replaced with $E \cup \{e\}$; to see that this is justified, observe that if the procedure gets through to line 22 at all, the call in line 16 must have actually computed $F = \text{opt}(G - \{e\})$; moreover, $F' < F$. This, however, means that e is enforced in (F', G), so for every future estimate $F'' \subseteq G$ with $F'' \leq F'$ we will have $e \in F''$, i.e.,

$$E \not\subseteq F'' \Leftrightarrow E \cup \{e\} \not\subseteq F'',$$

and the invariant is guaranteed.

It should be mentioned that the estimate F_e plugged into the recursive call in line 16 may be worse than some estimates computed during the **while** loop. The important property, however, is that F_e is at least as good as the original F we started with.

Toward the recurrence. The reader might wonder whether this algorithm really matches the rough idea we described. For example, we promised to make D as large as $\lceil c|\mathcal{D}(F, G)|\rceil$. This holds if $E = \emptyset$, because in this case, after the **while** loop,

$$|D| \geq \lceil c|G - E|\rceil = \lceil c|G|\rceil \geq \lceil c|\mathcal{D}(F, G)|\rceil,$$

but if $|E| > 0$, we might end up with a much smaller D. In this case, however, there are elements in $\mathcal{D}(F, G)$ (especially the ones in E) which are actually enforced in the sense that every estimate containing E has to contain these elements as well, and we should no longer consider such elements as free in a call with parameters E, F, and G. The following definition for triples takes care of that; it mimics Definition 4.1 for pairs.

DEFINITION 4.7. *For $E \subseteq F \subseteq G$, $e \in G - E$ is* enforced *in (E, F, G) if $F < \text{opt}(E, G - \{e\}) := \min_{<}\{F' \mid E \subseteq F' \subseteq G - \{e\}\}$ and e is* free *otherwise. The free elements form the* domain *of (E, F, G), i.e.,*

$$\mathcal{D}(E, F, G) := \{e \in G - E \mid \exists F' \leq F, E \subseteq F' \subseteq G - \{e\}\}.$$

The dimension *of (E, F, G) is the size of its domain, i.e.,*

$$\dim(E, F, G) := |\mathcal{D}(E, F, G)|.$$

This ensures that the domain contains exactly the elements which may potentially end up in D and guarantees that $|D| \geq \lceil c|\mathcal{D}(E, F, G)|\rceil$ after the **while** loop.

The main recurrence. To bound the expected performance of AOP, we can set up a recurrence which will look similar to the one that holds for AOP_SD. However, it will include an additional term for the effort that is necessary to find the set D; even worse, it will depend on two parameters rather than one, which makes it somewhat harder to solve. Nevertheless, it basically behaves like the one-parameter version and we will be able to establish a similar bound.

For fixed H and $m \geq k$, let $T(m, k)$ be the worst-case expected number of oracle queries performed in a call to AOP(E, F, G) with *size* $|G - E| = m$ and dimension $\dim(E, F, G) = k$, $G \subseteq H$. For a statement A let χ_A be 1 if A holds and 0 otherwise. We get $T(0, 0) = 1$ and for $m > 0$ we have the following bound.

THEOREM 4.8.

$$T(m, k) \leq \sum_{i=0}^{\lceil cm\rceil - 1} T(i, \min(i, k)) + \left[T(m-1, k-1) + 1 + \frac{1}{\lceil cm\rceil}\sum_{i=1}^{\lceil cm\rceil} T(m-1, k-i)\right]\chi_{k \geq \lceil cm\rceil}.$$

Proof. The arguments are basically the ones of Theorem 4.3, so we omit some details. Again, T is monotone in k; in the worst case, the **while** loop may be executed once with every value of $i := |G - E'|$ between 0 and $\lceil cm \rceil - 1$, which gives the first term. If $k < \lceil cm \rceil$, the algorithm will not be able to find the required number of free elements and thus cannot get beyond the **while** loop. Otherwise, the triple processed in line 16 has size $m-1$ and dimension at most $k-1$ (e is no longer free), which gives the $T(m-1, k-1)$ term. One more query may be necessary in line 19. Finally, the last term bounds the expected effort in line 22, averaged over the random choice in line 15 (let $G - E' = e_1, \ldots, e_l, l \geq \lceil cm \rceil$, ordered in such a way that

$$\text{opt}(E, G - \{e_1\}) \geq \cdots \geq \text{opt}(E, G - \{e_l\}),$$

and observe that $e_1, \ldots, e_i$ are no longer free in $(E \cup \{e\}, F', G)$ if $e = e_i$ is chosen in line 15). □

Solving the recurrence. It turns out that the dependence of $T(m, k)$ on m is quasi-polynomial, while the major contribution comes from k. Let us define two auxiliary functions:

$$f(k) = f(k-1) + \frac{1}{\lceil ck \rceil} \sum_{i=1}^{\lceil ck \rceil} f(k-i),$$

$$g(m) = g(m-1) + \sum_{i=0}^{\lceil cm \rceil - 1} g(i)$$

with $f(0) = g(0) = 1$.

LEMMA 4.9.

$$T(m, k) \leq f(k)g(m).$$

Proof. We proceed by induction on m. For $m = 0$ we have equality, so assume $m > 0$. Then

$$\begin{aligned}
T(m,k) &\leq \sum_{i=0}^{\lceil cm \rceil - 1} f(\min(i,k))g(i) \\
&\quad + \left[f(k-1)g(m-1) + 1 + \frac{1}{\lceil cm \rceil} \sum_{i=1}^{\lceil cm \rceil} f(k-i)g(m-1) \right] \chi_{k \geq \lceil cm \rceil} \\
&\leq f(k) \sum_{i=0}^{\lceil cm \rceil - 1} g(i) - \chi_{k>0} \\
&\quad + g(m-1) \left[f(k-1) + \frac{1}{\lceil ck \rceil} \sum_{i=1}^{\lceil ck \rceil} f(k-i) \right] \chi_{k \geq \lceil cm \rceil} + \chi_{k \geq \lceil cm \rceil} \\
&= f(k)[g(m) - g(m-1)] + g(m-1)f(k)\chi_{k \geq \lceil cm \rceil} + \chi_{k \geq \lceil cm \rceil} - \chi_{k>0} \\
&= f(k)g(m) - g(m-1)[f(k) - f(k)\chi_{k \geq \lceil cm \rceil}] + \chi_{k \geq \lceil cm \rceil} - \chi_{k>0} \\
&\leq f(k)g(m) - g(0)[f(0) - f(0)\chi_{k \geq \lceil cm \rceil}] + \chi_{k \geq \lceil cm \rceil} - \chi_{k>0} \\
&\leq f(k)g(m). \qquad \square
\end{aligned}$$

It remains to bound $f(k)$ and $g(m)$.

LEMMA 4.10.

$$\begin{aligned}
g(m) &\leq 3m^{\log_{1/c} m + 1} - 1 \text{ for } m > 0, \\
f(k) &\leq e^{2\sqrt{k/c}}.
\end{aligned}$$

Proof. The function $g(m)$ is monotone, so we can argue that $g(m) \leq g(m-1) + \lceil cm \rceil g(\lceil cm \rceil - 1)$, which by expanding $g(m-1)$ yields

$$g(m) \leq \sum_{i=1}^{m} \lceil ci \rceil g(\lceil ci \rceil - 1) + g(0) \leq m \lceil cm \rceil g(\lceil cm \rceil - 1) + 1 \leq m^2 g(\lceil cm \rceil - 1) + 1.$$

Now we claim that for $m > 0$, $g(m) \leq 3m^{\log_{1/c} m + 1} - 1$, which holds for $m = 1$, and inductively we obtain

$$g(m) \leq m^2 [3(cm)^{\log_{1/c} cm + 1} - 1] = 3m^2 m^{\log_{1/c} m - 1} - m^2 \leq 3m^{\log_{1/c} m + 1} - 1.$$

$f(k)$ can be majorized by the function $t(k)$ satisfying $t(0) = 1$ and

$$t(k) = t(k-1) + \frac{1}{ck} \sum_{i=1}^{k} t(k-i).$$

By applying the method of generating functions [8, §7] it is possible to compute $t(k)$ explicitly. One obtains

$$t(k) = \sum_{i=0}^{k} \binom{k}{i} \frac{(1/c)^i}{i!},$$

a generalization of the bound in Lemma 4.4. The formula can also be verified easily by induction, using standard binomial coefficient identities as can be found in [8, §5]. As in Corollary 4.5 we obtain from this the desired estimate

$$t(k) \leq e^{2\sqrt{k/c}}. \qquad \square$$

It takes more effort to show that $t(k) = \Theta(e^{2\sqrt{k/c}}/k^{3/4})$ (e.g., by using the saddle point method [9, p. 74 ff.]), so our simple estimate is tight up to a $k^{3/4}$ factor.

We have proved that the expected number of oracle queries performed by algorithm AOP, when called on a triple (E, F, G) of size m and dimension k, is bounded by

$$T(m, k) \leq 3e^{2\sqrt{k/c}} m^{\log_{1/c} m + 1}$$

for any fixed $0 < c < 1$. Setting $z := 1/c - 1$ and rewriting the expression shows that

$$T(n, n) \leq 3ne^{2\sqrt{n+zn} + \ln^2 n / \ln(1+z)}$$

queries are sufficient on the average to solve an AOP on a set H with n elements. By easy calculation, we see that the value z_0 minimizing this bound satisfies

$$z_0 = \frac{\ln n}{\sqrt[4]{n}}(1 + o(1)),$$

so a reasonable choice for z is $z = \ln n / \sqrt[4]{n}$. Exploiting the fact that $\ln(1+z) \geq z - z^2/2$ for positive z gives

$$\frac{\ln^2 n}{\ln(1+z)} \leq \frac{\ln^2 n}{z - z^2/2} = \frac{\ln^2 n}{z} + \frac{\ln^2 n}{2 - z} \leq \sqrt[4]{n} \ln n + \ln^2 n.$$

This derivation holds for $z \leq 1$; in case $z > 1$, the inequality follows directly. From the mean value theorem we get

$$\sqrt{n+zn} \leq \sqrt{n} + \frac{z}{2}\sqrt{n} \leq \sqrt{n} + \frac{1}{2}\sqrt[4]{n}\ln n.$$

This leads to an overall upper bound of

$$T(n,n) \leq 3ne^{2\sqrt{n}+2\sqrt[4]{n}\ln n+\ln^2 n}.$$

THEOREM 4.11. *An AOP $(H, <, \Phi)$ with $|H| = n$ can be solved with an expected number of no more than*

$$e^{2\sqrt{n}+2\sqrt[4]{n}\ln n+O(\ln^2 n)}$$

oracle queries.

5. Discussion. We have given an $e^{(2+o(1))\sqrt{n}}$ expected time randomized algorithm for AOPs on an n-element set H; the algorithm applies to the minimum spanning ball problem for n points in d-space and to the problem of computing the distance between two d-dimensional n-vertex (or n-facet) convex polyhedra, and for both problems we obtain the first subexponential combinatorial bounds in d. As in the case of linear programming one can obtain a bound of $O(d^2n + e^{O(\sqrt{d\log d})})$ for both problems by combining our result with the ones of [18] and [2].

Recently, Ludwig [16] has shown that the problem of finding optimal strategies for *simple stochastic games* [5] allows a subexponential solution (where he implicitly proved that the problem can be formulated as an AOP, which allows direct application of the procedure AOP_SD from §4.2); unlike the AOPs we discussed here, the simple stochastic game problem is of a more combinatorial rather than geometric flavor. In general, however, it seems that local optimization occurs most naturally in geometric settings.

To determine the randomized complexity of AOPs is itself a challenging problem. It seems that to substantially improve on the bound given here, one would have to come up with a method that does not rely on the concept of the dimension of an estimate as a measure of progress during the algorithm. Any better upper bound would immediately imply better algorithms for the "small" instances of the problems we have discussed, and we believe that the ideas behind such progress would carry over to the "large" instances. On the other hand, it is quite possible that there is a lower bound for AOPs that is in the range of the upper bound given here; however, so far we have not been able to establish any nontrivial lower bound. A natural time class to consider when thinking about upper or lower bounds seems to be $n^{\log^r n}$ for constant r.

Acknowledgments. The author thanks an anonymous referee for comments that helped to improve the presentation. Special thanks are extended to Emo Welzl for many discussions and for reading several versions of the manuscript.

REFERENCES

[1] K. L. CLARKSON, *Linear programming in $O(n3^{d^2})$*, Inform. Process. Lett., 22 (1986), pp. 21–24.

[2] ———, *Las Vegas algorithms for linear programming when the dimension is small*, 1989, manuscript.

[3] ———, *New applications of random sampling in computational geometry*, Discrete Comput. Geom., 2 (1987), pp. 195–222.

[4] B. CHAZELLE AND J. MATOUŠEK, *On linear-time deterministic algorithms for optimization problems in fixed dimension*, in Proc. 4th ACM-SIAM Symposium on Discrete Algorithms (SODA), Association for Computing Machinery, New York, NY, and Society for Industrial and Applied Mathematics, Philadelphia, PA, 1993, pp. 281–290.

[5] A. CONDON, *The complexity of stochastic games*, Inform. and Comput., 96 (1992), pp. 203–224.
[6] G. B. DANTZIG, *Maximization of a linear function of variables subject to linear inequalities*, in Activity Analysis of Production and Allocation, T. C. Koopman, ed., Cowles Commission Monograph 13, John Wiley, New York, 1951, pp. 339–347.
[7] M. E. DYER, *On a multidimensional search technique and its application to the euclidean one-center problem*, SIAM J. Comput., 15 (1986), pp. 725–738.
[8] R. L. GRAHAM, D. E. KNUTH, AND O. PATASHNIK, *Concrete Mathematics*, Addison–Wesley, Reading, MA, 1989.
[9] D. H. GREENE AND D. E. KNUTH, *Mathematics for the Analysis of Algorithms*, Birkhäuser, Boston, MA, 1981.
[10] M. GRÖTSCHEL, L. LOVÁSZ, AND A. SCHRIJVER, *Geometric Algorithms and Combinatorial Optimization*, Springer-Verlag, Berlin, 1988.
[11] G. KALAI, *A subexponential randomized simplex algorithm*, in Proc. 24th Annual ACM Symposium on Theory of Computing (STOC), Association for Computing Machinery, New York, NY, 1992, pp. 475–482.
[12] N. KARMARKAR, *A new polynomial-time algorithm for linear programming*, Combinatorica, 4 (1984), pp. 373–395.
[13] L. G. KHACHIYAN, *Polynomial algorithm in linear programming*, U.S.S.R. Comput. Math. and Math. Phys., 20 (1980), pp. 53–72.
[14] V. KLEE AND P. KLEINSCHMIDT, *The d-step conjecture and its relatives*, Math. Oper. Res., 12 (1987), pp. 718–755.
[15] V. KLEE AND G. J. MINTY, *How good is the simplex algorithm*, in Inequalities III, O. Shisha, ed., Academic Press, New York, NY, 1972, pp. 159–175.
[16] W. LUDWIG, *A subexponential randomized algorithm for the simple stochastic game problem*, Inform. and Comput., to appear.
[17] N. MEGIDDO, *Linear programming in linear time when the dimension is fixed*, J. Assoc. Comput. Mach., 31 (1984), pp. 114–127.
[18] J. MATOUŠEK, M. SHARIR, AND E. WELZL, *A subexponential bound for linear programming*, in Proc. 8th Annual Symposium on Computational Geometry, Association for Computing Machinery, New York, NY, 1992, pp. 1–8; Algorithmica, to appear.
[19] V. T. RAJAN, *Optimality of the Delaunay triangulation in* $\mathbb{R}^d$, in Proc. 7th Annual Symposium on Computational Geometry, Association for Computing Machinery, New York, NY, 1991, pp. 357–363.
[20] A. SCHRIJVER, *Theory of linear and integer programming*, John Wiley, New York, 1986.
[21] R. SEIDEL, *Low-dimensional linear programming and convex hulls made easy*, Discrete Comput. Geom., 6 (1991), pp. 423–434.
[22] M. SHARIR AND E. WELZL, *A combinatorial bound for linear programming and related problems*, in Proc. 9th Annual Symposium on Theoretical Aspects of Computer Science (STACS), Lecture Notes in Computer Science 577, Springer-Verlag, Berlin, 1992, pp. 569–579.
[23] K. SEKITANI AND Y. YAMAMOTO, *A recursive algorithm for finding the minimum norm point in a polytope and a pair of closest points in two polytopes*, Math. Programming, 61 (1993), pp. 233–249.
[24] E. WELZL, *Smallest enclosing disks (balls and ellipsoids)*, in New Results and New Trends in Computer Science, H. Maurer ed., Lecture Notes in Computer Science 555, Springer-Verlag, Berlin, 1991, pp. 359–370.
[25] P. WOLFE, *The simplex method for quadratic programming*, Econometrica, 27 (1959), pp. 382–398.

SIAM J. COMPUT.
Vol. 24, No. 5, pp. 1036–1050, October 1995

006

RANDOMNESS-OPTIMAL UNIQUE ELEMENT ISOLATION WITH APPLICATIONS TO PERFECT MATCHING AND RELATED PROBLEMS*

SURESH CHARI†, PANKAJ ROHATGI‡, AND ARAVIND SRINIVASAN§

Abstract. In this paper, we precisely characterize the *randomness complexity* of the unique element isolation problem, a crucial step in the *RNC* algorithm for perfect matching by Mulmuley, Vazirani, and Vazirani [*Combinatorica*, 7 (1987), pp. 105–113] and in several other applications. Given a set S and an unknown family $\mathcal{F} \subseteq 2^S$ with $|\mathcal{F}| \leq Z$, we present a scheme for assigning polynomially bounded weights to the elements of S using only $O(\log Z + \log |S|)$ random bits, such that the minimum weight set in $\mathcal{F}$ is unique with high probability. This generalizes the solution of Mulmuley, Vazirani, and Vazirani, who use $O(S \log S)$ bits, independent of Z. We also provide a matching lower bound for the randomness complexity of this problem. The new weight assignment scheme yields a randomness-efficient RNC^2 algorithm for perfect matching which uses $O(\log Z + \log n)$ random bits, where Z is any *given* upper bound on the number of perfect matchings in the input graph. This generalizes the result of Grigoriev and Karpinski [*Proc. IEEE Symposium on Foundations of Computer Science*, 1987, pp. 166–172], who present an NC^3 algorithm when Z is polynomial and improves the running time in this case. The worst-case randomness complexity of our algorithm is $O(n \log(m/n))$ random bits improving on the previous bound of $O(m \log n)$. Our scheme also gives randomness-efficient solutions for several problems where unique element isolation is used, such as *RNC* algorithms for variants of matching and basic problems on linear matroids. We obtain a randomness-efficient random reduction from SAT to USAT, the language of uniquely satisfiable formulas, which can be derandomized in the case of languages in Few$P$ to yield new proofs of the results Few$P \subseteq \oplus P$ and Few$P \subseteq C_= P$.

Key words. parallel algorithms, probabilistic algorithms, matching, matroids, random reductions

AMS subject classifications. 68Q20, 68Q25, 68Q15

1. Introduction.

Given a graph G, a *matching* in G is a subset of the edges such that no two edges in the subset are incident on the same vertex. A *maximum matching* in G is a matching in G of maximum cardinality and a *perfect matching* is a special type of maximum matching in which each vertex is covered by an edge, i.e., each vertex is *matched*. Finding a perfect matching, if any, in a graph is one of the fundamental problems in combinatorial optimization. While its sequential complexity has been well studied (Lovász and Plummer [20]), a big open question in the theory of parallel algorithms is whether a perfect matching can be found or even detected in *NC*. Because of its importance, considerable effort has been devoted toward developing parallel algorithms for the perfect matching problem. For instance, sublinear time parallel algorithms for general graphs (Goldberg, Plotkin, and Vaidya [10], Vaidya [32], and Grover [13]) and bipartite graphs (Goldberg et al. [9]) are known. *NC* algorithms have also been developed for special classes of graphs such as co-comparability graphs (Kozen, Vazirani, and Vazirani [18]), strongly chordal graphs (Dahlhaus and Karpinski [7]), graphs with polynomially many perfect matchings (Grigoriev and Karpinski [11] and Grigoriev, Karpinski, and Singer [12], and planar bipartite graphs (Miller and Naor [21]). A very successful approach for the general problem has been the use of randomness; both

*Received by the editors June 15, 1993; accepted for publication (in revised form) April 21, 1994. A preliminary version of this paper was presented at the 25th Annual ACM Symposium on the Theory of Computing.

†Department of Computer Science, Cornell University, Ithaca, New York 14853. The research of this author was supported in part by National Science Foundation grant CCR-9123730.

‡Department of Computer Science, Cornell University, Ithaca, New York 14853. The research of this author was supported in part by National Science Foundation grant CCR-9123730, an IBM Graduate Fellowship, the United States Army Research Office through the Army Center of Excellence for Symbolic Methods in Algorithmic Mathematics (ACSyAM), and Mathematical Sciences Institute of Cornell University contract DAAL03-91-C-0027.

§Department of Computer Science, Cornell University, Ithaca, New York 14853. The research of this author was supported in part by National Science Foundation Presidential Young Investigator award CCR–89–96272 with matching support from United Parcel Service and Sun Microsystems, and by an IBM Graduate Fellowship.

Monte Carlo *RNC* algorithms (Karp, Upfal, and Wigderson [15] and Mulmuley, Vazirani, and Vazirani [22]) and Las Vegas *RNC* algorithms (Karloff [14]) have been developed.

In this paper, we investigate the *parallel randomness complexity* of perfect matching and related problems, i.e., the amount of randomness required to solve them in parallel. For perfect matching we present an RNC^2 algorithm which uses $O(\log Z + \log n)$ random bits, where Z is any *given* upper bound on $|\mathrm{Mat}(G)|$, the number of perfect matchings in the graph G. This improves and generalizes the result of Grigoriev and Karpinski [11], who give an NC^3 algorithm when Z is polynomially bounded. Since a graph on n vertices with m edges can have at most $(2m/n)^n$ perfect matchings, the worst-case randomness complexity of our algorithm is $O(n \log(m/n))$, which improves on the previous bound of $O(m \log n)$ from Mulmuley, Vazirani, and Vazirani [22]. Thus, by linking the randomness complexity of this problem to the number of perfect matchings, we unify and generalize previous results while also improving on them. In special cases, e.g., $K_{3,3}$-free graphs, the number of perfect matchings in the graph can be computed in NC^2 (see, for example, Vazirani [34] and also the work of Kastelyn [16] and Little [19]); in general, if no good upper bound on $|\mathrm{Mat}(G)|$ is known, our results yield an RNC^3 algorithm for finding a perfect matching which uses at most $O(\log |\mathrm{Mat}(G)|)$ random bits with high probability if the input graph has at least one matching. This is a significant reduction if $0 < |\mathrm{Mat}(G)| \ll (2m/n)^n$. All of these results are obtained from a *randomness-optimal* generalization of the isolating lemma, a tool used to isolate a perfect matching in the *RNC* algorithm of Mulmuley, Vazirani, and Vazirani [22]. Since this abstract and powerful tool has several applications, such as those to basic problems on linearly representable matroids, variants of matching, and random reductions in structural complexity theory, our generalization results in randomness-efficiency in these settings as well.

Given a set $S = \{x_1, x_2, \ldots, x_N\}$ and an *unknown* family $\mathcal{F} \subseteq 2^S$, the isolating lemma of [22] shows that if the x_i's are *independently* assigned weights, uniformly at random from the range $\{1, 2, \ldots, 2N\}$, then with probability at least 1/2, the minimum weight set in $\mathcal{F}$ is *unique*. This weight assignment scheme clearly requires $\Theta(N \log N)$ random bits, and a question left open in the original paper was whether the assumption that the weights be assigned independently is necessary. Our scheme generalizes this as follows: given an upper bound Z on $|\mathcal{F}|$, we assign polynomially bounded weights to the x_i's using only $O(\log Z + \log N)$ random bits and achieve the same result. In the worst case ($Z = 2^N$), our scheme needs $O(N)$ random bits as compared to $\Theta(N \log N)$; more importantly, for smaller Z, we get much better randomness complexity. Also, since even in the worst case we assign polynomial weights to N different random variables using $O(N)$ random bits, the weight assignments are *not* independent, thus settling the open question of [22]. Our scheme provides an explicit construction of a collection of $(NZ)^{O(1)}$ weight assignments such that, for every family $\mathcal{F}$ of size at most Z, at least one (in fact, at least 50%) of these assignments makes the minimum weight subset in $\mathcal{F}$ unique. Such a collection of size $O(NZ)$ is guaranteed to exist by applying Adleman's technique [1] to the original isolating lemma. Thus, this is one of the few instances where Adleman's result can be made constructive with only a polynomial blowup in size. Furthermore, we also show that our weight assignment scheme is optimal by proving a matching lower bound for this generalization, i.e., any scheme which assigns polynomial weights to the x_i's must use $\Omega(\log Z + \log N)$ random bits even to achieve isolation with only *nonzero* probability.

This randomness-efficient generalization of the isolation process can be plugged directly into the settings where the original isolating lemma has been used to obtain randomness efficiency. In addition to the new algorithm for perfect matching we get randomness-efficient algorithms for variants of perfect matching, such as exact matching and maximum match-

ing. We get RNC^2 algorithms for exact and maximum matching which use $O(\log Z + \log n)$ random bits, where Z is a *given* upper bound on the number of exact matchings and maximum matchings, respectively. Matroid intersection and matroid matching are generalizations of bipartite and general graph matching, and the isolating lemma has been used to obtain RNC^2 algorithms for these basic problems on linearly representable matroids (Narayanan, Saran, and Vazirani [23]). Here, by obtaining good bounds on the size of the family on which the isolating lemma operates, we obtain significant savings in the number of random bits required. The randomness complexities of our algorithms for matroids, when specialized to the case of matching, yield precisely our results on matching. In particular, we give NC^2 algorithms for these problems on matroids under certain restrictions, which, in the case of graph matching, correspond exactly to a polynomial bound on the number of perfect matchings. This generalizes the results of [11] to matroids and also generalizes a result of Tiwari [29].

Another important application of the isolating lemma is to obtain an alternative to the Valiant and Vazirani random reduction [33] from SAT to USAT. Our generalization yields a new random reduction whose randomness complexity is logarithmic in the number of satisfying assignments. The worst-case complexity matches the best-known bound of $O(n)$ (Tarui [27]) and directly gives new proofs of the results that Few$P$, where the number of satisfying assignments is polynomially bounded, is contained in $\oplus P$ (Cai and Hemachandra [3]) and $C_{=}P$ (Köbler et al. [17]).

This paper is organized as follows: In §2 we describe the weight assignment scheme and show that it has optimal randomness complexity. Section 3 lists the applications of the new tool to perfect matching and its variants. Applications to problems on linearly representable matroids and random reductions in structural complexity are listed in §4.

2. The generalized isolating lemma. In this section we present our randomness-efficient generalization of the isolating lemma and show its optimality by proving a matching randomness complexity lower bound for this problem. We start with a formal statement of the isolating lemma.

DEFINITION. *A* set system $(S, \mathcal{F})$ *consists of a finite set* $S = \{x_1, \ldots, x_N\}$ *and a family* $\mathcal{F}$ *of subsets of* S. *A weight assignment* $\vec{w} = \langle w_1, \ldots, w_N \rangle$ *to the elements* $x_1, \ldots, x_N$, *extends naturally to sets in* $\mathcal{F}$ *with* $\vec{w}(S_j) = \sum_{x_i \in S_j} w_i$.

The crux of the RNC^2 algorithm of Mulmuley, Vazirani, and Vazirani is the following probabilistic tool.

LEMMA 2.1 (isolating lemma [22]). *Let* $(S, \mathcal{F})$ *be any set system. If the elements* x_i *of* S *are assigned random weights* uniformly and independently *from* $\{1, 2, \ldots, 2N\}$ *then, with probability at least* $1/2$, *there will be a* unique *minimum weight set in* $\mathcal{F}$. *Two features make the isolating lemma widely applicable. First, the isolation process works for arbitrary and unknown families* $\mathcal{F}$, *and second, isolation is achieved by assigning only polynomial weights. For example, in the bipartite perfect matching problem where* S *is the set of edges and* $\mathcal{F}$ *is the collection of perfect matchings, the first feature allows isolation to be done in RNC without any knowledge of the perfect matchings, and the second ensures that a matching can be found by inverting a matrix with* polynomial-*sized entries* [22].

2.1. New isolation scheme. We prove a randomness-efficient generalization of the isolating lemma which also assigns polynomial weights and depends only on any given upper bound Z on the size of the *unknown* family $\mathcal{F}$. The motivation for this generalization is that, in several settings where the isolating lemma has been used to obtain randomized algorithms in the general case, we have deterministic solutions when the number of solutions is small [11], [29], [3]. These deterministic solutions have been obtained by techniques that are

specialized to each case, and our aim is to obtain them by carefully parametrizing the randomness complexity of the algorithms for the general case. To do this we prove the following generalization.

LEMMA 2.2 (Generalized Isolating Lemma). *Let $(S, \mathcal{F})$ be any set system and let Z be a* given *upper bound on the size of the unknown family $\mathcal{F}$. There is a simple scheme which uses $O(\log Z + \log N)$ random bits to assign integer weights to the x_i's in the range $[0, N^7)$ such that, with probability at least $1/4$, there is a* unique *minimum weight set in $\mathcal{F}$.*

Proof. We outline a four-step process for assigning weights to the x_i's and prove that this scheme has the desired properties. At step j we assign an intermediate weight $\vec{w}^{(j)}$. Only Steps 2 and 4 are randomized and together use $O(\log Z + \log n)$ random bits.

Since $\mathcal{F}$ is unknown, we deterministically assign very large weights in Step 1 so that every set in $\mathcal{F}$ gets a *distinct* weight.

Step 1. For each i, set $w_i^{(1)} = 2^i$.

Under $\vec{w}^{(1)}$, sets in $\mathcal{F}$ have *distinct* weights in $\{1, 2, \ldots, 2^{N+1}\}$. Clearly, if $Z \ll 2^{N+1}$, the same property should be obtainable with much lower weights. Since $\mathcal{F}$ is unknown, a deterministic strategy for reducing weights may fail; instead, we use a randomized strategy.

Step 2. Choose m uniformly at random from $\{1, 2, \ldots, (2NZ^2)^2\}$. For each i, define $w_i^{(2)} = w_i^{(1)} \bmod m$.

Step 2 requires $O(\log Z + \log N)$ random bits. Under $\vec{w}^{(2)}$, the Z or fewer sets in $\mathcal{F}$ have weights in the interval $[0, \min(N(2NZ^2)^2, 2^{N+1})]$, which is a big improvement for small values of Z. We now claim that, with good probability, these weights are also *distinct.*

CLAIM 1. *With probability at least $1/2$,* all *sets in $\mathcal{F}$ have* distinct *weights under $\vec{w}^{(2)}$.*

Proof. Suppose $\mathcal{F} = \{S_1, \ldots, S_k\}$, where $k \leq Z$. Consider the (unknown) integer

$$I = \prod_{1 \leq i < j \leq k} (\vec{w}^{(1)}(S_i) - \vec{w}^{(1)}(S_j)).$$

Clearly the properties of $\vec{w}^{(1)}$ ensure that $I \neq 0$ and $|I| < 2^{2NZ^2}$. The following number-theoretic proposition together with the Chinese remainder theorem establishes that, when m is chosen uniformly from $\{1, 2, \ldots, (2NZ^2)^2\}$, the probability that $m \nmid I$ is at least $1/2$. Several versions of this proposition appear in the literature; this version is from [31] and the constant 89 has been improved to 3 by Thrash [28].

PROPOSITION 2.3. *Let $L \geq 89$ and let S be any subset of $\{1, \ldots, L^2\}$ such that $|S| \geq \frac{1}{2}L^2$. Then, the least common multiple of the elements of S exceeds 2^L.*

We claim that if $m \nmid I$, then all sets in $\mathcal{F}$ have *distinct* weights under $\vec{w}^{(2)}$. If not, then $\vec{w}^{(2)}(S_i) = \vec{w}^{(2)}(S_j)$ for some $i < j$. Then we have $m \mid (\vec{w}^{(1)}(S_i) - \vec{w}^{(1)}(S_j))$, which contradicts the fact that $m \nmid I$. □

Remark. We say that Step 2 *succeeds* if sets in $\mathcal{F}$ get distinct weights under $\vec{w}^{(2)}$. The success probability of this step can be boosted to any constant less than 1, since it follows from the results of de Bruijn [8] that for any fixed $p < 1$, there exist constants c_1 and c_2 such that if m is picked uniformly at random from $\{1, 2, \ldots, N^{c_1}Z^{c_2}\}$ in Step 2, then Step 2 will succeed with probability at least p.

Note that if $Z \leq N^c$ then $w_i^{(2)} \leq N^{4c+3}$, and with probability at least $1/2$ *all* sets in $\mathcal{F}$ have *distinct* weights. Later, we use this strong condition to design an NC^2 algorithm to find *all* perfect matchings in graphs with at most $N^{O(1)}$ perfect matchings. For larger Z, the weights are still too big and in Steps 3 and 4 we reduce them further. Step 2 assigns q-bit weights to the x_i's, where $q = \min(N, \log m) \leq \min(N, 4\log Z + 2\log(2N))$. Let $t = \lceil q/\log N \rceil$.

Step 3. For each i write $w_i^{(2)}$ as q-bit number. Split these bits into t blocks of size $\log N$ bits each as shown. Let $b_{i,j}$ be the number in $[0, N-1]$ formed by the bits in block j as

follows:

$w_i^{(2)}$: $\overbrace{\boxed{b_{i,t-1}}\ \cdots\ \boxed{b_{i,1}}\ \boxed{b_{i,0}}}^{q}$. $\underbrace{\qquad\qquad}_{\log N}$

Let $w_i^{(3)}$ be the *linear form* $\sum_{j=0}^{t-1} b_{i,j} \cdot y_j$ over the variables $y_0, \ldots, y_{t-1}$.

CLAIM 2. *If Step 2 succeeds then the linear forms* $\vec{w}^{(3)}(S_j)$, *where* $S_j \in \mathcal{F}$, *are all* distinct.

Proof. Assume that Step 2 succeeds, i.e., that all the weights $\vec{w}^{(2)}(S_j)$ are *distinct*, where $S_j \in \mathcal{F}$. Note that each $w_i^{(3)}$ *evaluated* at $y_k = 2^{k \log N}$, $0 \le k \le t-1$, is *exactly* $w_i^{(2)}$. This implies that each $\vec{w}^{(3)}(S_j)$ evaluates to the *distinct* value $\vec{w}^{(2)}(S_j)$ at $y_k = 2^{k \log N}, 0 \le k \le t-1$, which implies that the forms $\vec{w}^{(3)}(S_j)$ must be distinct. □

Note that each $\vec{w}^{(3)}(S_j)$ is a linear form with coefficients in the range $[0, N(N-1)]$. We will use this property in a crucial way in the analysis of the next and final step.

Step 4. Choose $r_0, \ldots, r_{t-1}$ uniformly and independently at random from $\{1, 2, \ldots, N^5\}$. For each i, set $w_i^{(4)}$ as the evaluation of $w_i^{(3)}$ at $y_k = r_k$, $0 \le k \le t-1$.

We claim that $\vec{w}^{(4)}$ achieves the requirements of the generalized isolating lemma. Clearly, Step 4 requires $5 \log N \times t = O(\log Z + \log N)$ random bits and, since Step 2 has a similar randomness complexity, the overall procedure requires $O(\log Z + \log N)$ random bits. It is easy to check that each $w_i^{(4)}$ is in $[0, N^7)$. Since Step 2 succeeds with probability at least $1/2$, by using $\mathcal{C} = \{\vec{w}^{(3)}(S_j) \mid S_j \in \mathcal{F}\}$ in the following proposition, we obtain that the weight assignment $\vec{w}^{(4)}$ achieves isolation with probability at least $1/4$.

PROPOSITION 2.4. *Let* $\mathcal{C}$ *be any collection of* distinct *linear forms over at most* t *variables* $\vec{y} = y_0, y_1, \ldots, y_{t-1}$ *with coefficients in* $\{0, 1, \ldots, N^2-1\}$. *Choose a random* $\vec{r} = r_0, \ldots, r_{t-1}$ *by assigning each* r_i *uniformly and independently from* $\{1, 2, \ldots, N^5\}$. *Then, in the assignment* $\vec{y} = \vec{r}$ *there will be a* unique *linear form with minimum value, with probability at least* $1/2$.

Proof of Proposition 2.4. Our proof parallels that of [22]; we define a variable y_i as *singular* under an assignment $\vec{r}$ to the variables $\vec{y}$ if there exist two minimum-valued linear forms in $\mathcal{C}$ under this assignment having different coefficients of y_i. Since all the linear forms in $\mathcal{C}$ are initially distinct, if two linear forms attain the minimum weight then there *must* exist some variable which is singular under this assignment.

For each y_i, we will upper bound the probability that it is *singular*. Fix an i and assume that the variables $\vec{r}(\bar{i}) = r_0, \ldots, r_{i-1}, r_{i+1}, \ldots, r_{l-1}$ have been assigned the values $\vec{a}(\bar{i}) = a_0, \ldots, a_{i-1}, a_{i+1}, \ldots, a_{N-1}$. Under this partial assignment the weight of each linear form is of the form $a + by_i$, where a is the *partial weight* of the linear form and the coefficient b is at most $N^2 - 1$. The family $\mathcal{C}$ can be partitioned into at most N^2 classes $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_{N^2-1}$, based on the coefficient of y_i, i.e., forms in $\mathcal{C}_j$ have j as the coefficient of y_i. By definition, y_i is *singular* under an assignment $\vec{r}$ if and only if there are two linear forms from two different classes in the partition which attain the minimum weight. Consider the probability that the choice of r_i makes y_i singular, conditioned on the partial assignment to the other variables. Let p_j be the minimum partial weight among all the forms in $\mathcal{C}_j$. Note that since all the forms in a particular class have the same coefficient of y_j in their weights, only the forms with minimum partial weight in each class can possibly attain the minimum overall weight. Thus the probability that r_i makes y_i *singular* is exactly the probability that the minimum-valued element in the list $\mathcal{D} = [p_0, p_1 + r_i, p_2 + 2r_i, \ldots, p_{N^2-1} + (N^2-1)r_i]$ is not unique. This is clearly bounded by the probability that under a random r_i the values of elements in $\mathcal{D}$ are not all distinct. For each pair of elements in $\mathcal{D}$, *at most one* choice of r_i makes their values equal.

Thus

$$\begin{aligned}\text{Prob}\,[\ y_i \text{ is singular} \mid \vec{r}(\bar{i}) = \vec{a}(\bar{i})\ ] &\le \text{Prob}\,[\ \exists l, m:\ p_l + l \times r_i = p_m + m \times r_i] \\ &\le \binom{N^2}{2} \times \text{Prob}\,[\ p_l + l \times r_i = p_m + m \times r_i\ ] \\ &\le \binom{N^2}{2} \times \frac{1}{N^5} = \frac{1}{2N}.\end{aligned}$$

Since r_i is chosen independently of the other variables $\vec{r}(\bar{i})$, the probability that the assignment makes y_i singular is at most $1/2N$. Since this holds for each variable and there are at most $t < N$ variables, we have

$$\begin{aligned}\text{Prob}\,[\ \text{minimum valued form in } \mathcal{C} \text{ is unique}\ ] &\ge 1 - \text{Prob}\,[\exists j\ \ y_j \text{ is singular}] \\ &\ge 1 - t \times \frac{1}{2N} > \frac{1}{2}. \qquad \square\end{aligned}$$

Since each of the four steps is elementary, the entire weight assignment scheme can be implemented in NC^1. Finally, we note that the generalized isolating lemma provides an *explicit construction* of $(NZ)^{O(1)}$ weight assignment functions for the ground set $S = \{x_1, x_2, \ldots, x_N\}$ such that, for any set system $(S, \mathcal{F})$ with $|\mathcal{F}| \le Z$, at least half of these functions will produce a unique minimum weight set in $\mathcal{F}$. For a given ground set S, the number of set systems $(S, \mathcal{F})$ with $|\mathcal{F}| \le Z$ is $\sum_{i=1}^{Z} \binom{2^N}{i}$; hence, the technique of Adleman [1] combined with the original isolating lemma shows *nonconstructively* that there exists a set of $O(\log(\sum_{i=1}^{Z} \binom{2^N}{i})) = O(NZ)$ weight assignment functions such that, for any set system $(S, \mathcal{F})$ with $|\mathcal{F}| \le Z$, at least *one* of these functions will produce a unique minimum weight set in $\mathcal{F}$.

2.2. A lower bound for the isolation problem. We establish a matching lower bound of $\Omega(\log Z + \log N)$ random bits on the randomness complexity of the generalized isolation process and, in fact, for the following weaker problem, thus showing that the randomness complexity of our weight assignment scheme is optimal to within a constant factor.

Generalized Isolation. Let $S = \{x_1, \ldots, x_N\}$ and let k be a constant. Assign weights to the elements of S in the range $[1, N^k]$ such that for any family $\mathcal{F} \subseteq 2^S$ of size at most Z, the minimum weight set in $\mathcal{F}$ is unique with *nonzero* probability.

First we prove a lower bound of $\Omega(\log Z)$ random bits for any randomized scheme for the above problem. Each path of such a randomized algorithm defines a weight assignment f to the x_i's. The following theorem shows that at least $t + 1$ different weight assignments are needed to achieve isolation in all possible families of size at most Z, where $t = \min(\lfloor \frac{Z}{2} \rfloor, \lfloor (2^N - 3)/N^{k+1} \rfloor)$. Hence any randomized scheme for generalized isolation must use $\Omega(\log t) = \Omega(\log Z)$ random bits.

THEOREM 2.5. *Let $f_1, f_2, \ldots, f_t$ be any collection of weight assignments that assign weights in the range $[1, B]$ to the elements of S. If $t \le \min(\frac{Z}{2}, (2^N - 3)/NB)$, then there exists a family $\mathcal{F}$ of subsets of S with $|\mathcal{F}| \le 2t \le Z$, such that the minimum weight subset of $\mathcal{F}$ is not unique under any of these assignments.*

Proof. Given any t weight assignments, we explicitly construct the family $\mathcal{F}$ as follows. First, for each f_i we compute the "histogram" of the weights of the subsets of S, i.e., for each nonempty subset X we place a mark above the weight $f_i(X)$ in the histogram for f_i; one distinct mark is made for each such subset X.

The weight of any subset is in the range $[1, NB]$. For each i initialize a pointer p_i to 1 in the histogram for f_i. The pointers p_i advance according to the following rules:

- If there are no marks on weight p_i in the histogram of f_i, then advance p_i to $p_i + 1$.
- If there is exactly one mark on weight p_i which corresponds to some subset X, remove the marks corresponding to X from *all* the histograms and advance p_i to $p_i + 1$.

If more than one pointer can advance, we choose one among them arbitrarily. We continue this process until the pointers cannot move any further. First we argue that not all marks can be removed, since every time a mark is removed one of the pointers advances, and hence the potential function $\Psi = \sum_i (p_i - 1)$ always increases. Since each p_i is bounded by $NB + 1$, Ψ is bounded by tNB. By assumption $tNB \leq 2^N - 3$, and hence when the pointers come to a halt, there are at least two marks corresponding to nonempty subsets X_{i1} and X_{i2} on weight p_i in the histogram of f_i for each i. The family $\mathcal{F} = \{X_{i1}, X_{i2} | \ 1 \leq i \leq t\}$ has size at most $2t \leq Z$, and for each i there are two nonempty minimum weight subsets X_{i1} and X_{i2} of minimum weight p_i under the assignment f_i. □

Also, when $B = n^{O(1)}$ and $t = o(N/\log N)$, we can construct two sets which have the same weight under all the assignments $f_1, f_2, \ldots, f_t$ as follows. Since f_1 maps the nonempty subsets of S to the range $[1, NB]$, there exists a family A_1 of nonempty subsets of S, all of which get the same weight under f_1 and are such that $|A_1| \geq (2^N - 1)/NB$. Similarly, there exists a family $A_2 \subseteq A_1$ with $|A_2| \geq (2^N - 1)/(NB)^2$, such that $f_i(X_1) = f_i(X_2)\ \forall X_1, X_2 \in A_2$ for $i = 1, 2$. Repeating this, we end up with a set A_t of nonempty subsets of S with $|A_t| \geq (2^N - 1)/(NB)^t$, such that the minimum weight element of A_t is not unique under any $f_1, f_2, \ldots, f_t$; since $(2^N - 1)/(NB)^t \geq 2$ for $t = o(N/\log N)$, our claim holds. Thus, any scheme for generalized isolation requires $\Omega(\log Z + \log N)$ random bits. Theorem 2.5 also gives similar lower bounds when the elements are assigned superpolynomial weights. In fact, we can prove the same lower bound when the assignments map subsets of S to an arbitrary linearly ordered set of size B, since the aforementioned proofs use no property of the integers other than their total ordering.

3. Applications to matching problems. The generalized isolating lemma immediately yields randomness-efficient algorithms for several problems related to matching. In this section, we consider only matchings in bipartite graphs; with more work, identical results can be obtained for matchings in general graphs [20], [22]. The failure probabilities of all our algorithms can be made inverse polynomial with only a constant blowup in the time and randomness complexities and a polynomial blowup in the number of processors using two point sampling techniques of Chor and Goldreich [5]. The new isolating lemma is a general tool for bounding the randomness complexity of a problem in terms of the number of its solutions. However, because of the large polynomial weights used by our lemma, algorithms based directly on the lemma usually suffer a processor penalty.

Notation. We denote a bipartite graph G by $G = (U, V, E)$, where $U = \{u_1, u_2, \ldots, u_{n/2}\}$, $V = \{v_1, v_2, \ldots, v_{n/2}\}$, and $|E| = m$. The determinant of a matrix A is denoted by $\det(A)$. The number of perfect matchings in G is denoted by $|\mathrm{Mat}(G)|$.

3.1. Algorithms for matching. To apply unique element isolation to perfect matching, we let the ground set be the set of edges and let the family of subsets be the set of perfect matchings of G as in [22]. By the generalized isolating lemma, given an upper bound Z on $|\mathrm{Mat}(G)|$, we can assign polynomially bounded weights to the edges of G using $O(\log Z + \log n)$ random bits, such that there is a unique perfect matching of minimum weight with good probability. We construct a matrix $M[1..n/2, 1..n/2]$ with

$$M[i, j] = \begin{cases} 2^{w_{ij}} & \text{if } (u_i, v_j) \in E, \\ 0 & \text{otherwise,} \end{cases}$$

where w_{ij} is the weight assigned to edge (u_i, v_j). A perfect matching in G can now be found, if one exists, by inverting M via the NC^2 algorithm of Csanky [6] as shown in [22]. Note

further that the additive $O(\log n)$ factor in the randomness complexity can be absorbed by a processor penalty. For any graph $G = (V, E)$, $|\mathrm{Mat}(G)| \leq \prod_{v \in V} \mathrm{degree}(v)$, which is at most $(2m/n)^n$, since the sum of the degrees of the vertices is $2m$. In the worst case, by letting $Z = (2m/n)^n$ we obtain a randomness complexity of $O(n \log(m/n))$, which improves the previous bound of $O(m \log n)$ [22]. Thus we get the following theorem.

THEOREM 3.1. *There is an RNC^2 algorithm for finding a perfect matching in a graph G (if one exists) which uses $O(\log Z + \log n)$ random bits given an upper bound Z on $|\mathrm{Mat}(G)|$. In the worst case this algorithm uses $O(n \log(m/n))$ random bits.*

A careful analysis of the proof of our isolating scheme, when applied to the worst case where $Z = (2m/n)^n$ and $m = \Theta(n^2)$, shows that the following algorithm also achieves the same randomness complexity: the edges of each vertex $u \in U$ are assigned weights *pairwise independently* in the range $[1, m^7]$.

Perfect matching is a special case of the *exact matching problem*: Given a graph G with edges colored red and blue arbitrarily and an integer k, an exact matching is a perfect matching of G with *exactly* k red edges (Papadimitriou and Yannakakis [26]). Even though the problem of finding an exact matching is not known to be in P, the isolating lemma has been used to solve it in RNC^2 [22]! By observing that isolation is needed only among the exact matchings, we can link the randomness complexity of this problem to a bound X on the number of exact matchings and *not* on the number of perfect matchings. Suppose we assign a nonnegative integral weight w_{ij} to every edge $(u_i, v_j) \in E$. Let y be an indeterminate; construct a symbolic matrix $M_y[1..n/2, 1..n/2]$ such that

$$M_y[i, j] = \begin{cases} y \cdot w_{i,j} & \text{if } (u_i, v_j) \in E \text{ and is colored red,} \\ w_{ij} & \text{if } (u_i, v_j) \in E \text{ and is colored blue,} \\ 0 & \text{otherwise.} \end{cases}$$

Then, it is easy to see that the contribution to the coefficient of y^k in $\det(M_y)$ comes precisely from the exact matchings in G and that if there is a unique minimum weight exact matching M^*, then its weight equals the highest power of 2, say p^*, which divides this coefficient of y^k. Furthermore, any edge can be tested for membership in M^* by increasing its weight by 1 and testing if the new value of p^* is higher than its old value. Hence, this problem can be solved by using our generalized isolating lemma over the ground set E and the (unknown) family of exact matchings of G. Since the determinant of a matrix in one variable can be computed in NC^2 (Borodin, Cook, and Pippenger [2]), we get the following theorem.

THEOREM 3.2. *Given an upper bound X on the number of exact matchings in a graph G, an exact matching (if any) in G can be found in RNC^2 using $O(\log X)$ random bits.*

The isolating lemma has also been used to find the matching of largest cardinality in bipartite graphs by a reduction to the perfect matching problem. However, the NC reductions that have been used earlier [22], [20] do not suffice to get randomness-efficient solutions since they cause a blowup in the number of solutions. Instead, we use the following reduction from maximum matching to exact matching. Given a graph $G = (V, E)$, we first construct two copies of G and color the edges in each of the copies red. Each vertex v in the first copy of G is joined to vertex v in the second copy by a blue edge to yield the resulting graph G_1. If m is the size of the largest matching in G then it can easily be seen that there is an exact matching with $2m$ red edges in G_1, but no exact matching with more than $2m$ red edges. Also, if Z is an upper bound on the number of maximum matchings in the graph G, the new graph has at most Z^2 exact matchings with $2m$ red edges. Also note that if G is bipartite then so is G_1. Thus, for maximum matching we first construct the new graph G' and the new bound Z^2, and run the above algorithm for exact matching with each possible value of m simultaneously in

parallel using the same random bits. We output the matching of largest cardinality that results in any of the parallel runs.

THEOREM 3.3. *There exists an RNC^2 algorithm for maximum matching that uses $O(\log Z + \log n)$ random bits, where Z is a known bound on the number of maximum matchings in G.*

3.2. Graphs with a polynomial number of perfect matchings. Grigoriev and Karpinski [11] consider the problem of finding all the perfect matchings in graphs with a polynomial bound on the number of perfect matchings. Using techniques from algebra, they give an NC^2 algorithm for this problem if the number of matchings is bounded by $\log^{0.5-\epsilon} n$, and an NC^3 algorithm when the bound is n^c for constant c. Since their algorithm for extracting a matching is limited by their procedure for detecting if a matching exists, they suggest that $\log^{0.5-\epsilon} n$ may be a limiting upper bound on the number of matchings for finding all (or even one) of the perfect matchings in NC^2 via parallel algebra. We present an NC^2 algorithm for the detection problem which provides enough information to find all matchings in NC^2 even when the number of matchings is polynomially bounded. In addition to the improvement in running time, our algorithm is considerably simpler than the algorithm of Grigoriev and Karpinski.

As before, we let S and $\mathcal{F}$ be E and $\mathrm{Mat}(G)$, respectively. Running Steps 1 and 2 of the generalized isolating scheme, we use $O(\log n)$ random bits to assign each edge a polynomially bounded weight such that *all* the perfect matchings of G have *distinct* weights with probability at least $1/2$. Let $w_{i,j}$ be the weight assigned to edge (u_i, v_j). Construct a matrix A with

$$A[i,j] = \begin{cases} 2^{2\times w_{ij}} & \text{if } (u_i, v_j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Since every matching has a distinct weight, $\det(A)$ gives information about *every* perfect matching as shown below.

Assume that Step 2 succeeds, i.e., all the perfect matchings indeed get distinct weights. If we denote the matrix obtained from M by removing its ith row and jth column by M_{ij}, then by definition,

$$(M^{-1})[j,i] = \frac{(-1)^{i+j}\det(M_{ij})}{\det(M)}.$$

Now, if there is a perfect matching of weight k containing the edge (u_i, v_j), then $\det(M_{ij})$ is of the form

$$\pm 2^{2\times(k-w_{ij})} + \sum_{\ell=0}^{k-1} a_\ell 2^{2\times(\ell-w_{ij})} + \sum_{\ell\geq k+1} a_\ell 2^{2\times(\ell-w_{ij})},$$

where each a_ℓ is in $\{-1, 0, 1\}$. Since

$$\left|\sum_{\ell=0}^{k-1} a_\ell 2^{2\times(\ell-w_{ij})}\right| < 2^{2\times(k-w_{ij})-1},$$

this is of the form

$$\pm 2^{2\times(k-w_{ij})} \pm n_0 \times 2^{2(k-w_{ij}+1)} \pm x \times 2^{2(k-w_{ij})-1},$$

where n_0 is a nonnegative integer and $0 \leq x < 1$. Similarly, if there is no perfect matching of weight k containing the edge (u_i, v_j), then $\det(M_{ij})$ is of the form

$$\pm n_0 \times 2^{2(k-w_{ij}+1)} \pm x \times 2^{2(k-w_{ij})-1}.$$

Hence, there is a perfect matching of weight k containing the edge (u_i, v_j) if and only if

$$|(M^{-1})[j,i] \cdot \det(M) \cdot 2^{2w_{ij}}| \pmod{2^{2(k-1)}}$$

is of the form $4n_1 \pm y$, where n_1 is an integer and $y = 1$ or 2. If there is no matching of weight k then $y = 0$ or 3. Since k is polynomially bounded, we can extract all matchings. Also, since we use $O(\log Z + \log n) = O(\log n)$ random bits, we can derandomize the reduction by searching over all points in the sample space in parallel to get the following theorem.

THEOREM 3.4. *There is an NC^2 algorithm for detecting if a perfect matching exists and for finding all perfect matchings in a graph G, when givren that $|\mathrm{Mat}(G)| \leq n^c$ for some constant c.*

We can obtain the same result for exact matching and maximum matching problems given a polynomial bound on the number of exact matchings and maximum matchings, respectively.

Grigoriev and Karpinski also give a Las Vegas RNC^3 algorithm for the problem of testing if a graph has at most n^c perfect matchings, which uses expected $O(n^{2c+8.5} \log n)$ random bits. Using the algorithm of Theorem 3.4 and extending ideas in [18] and [14], we can improve on the running time and substantially on the number of random bits used. We will use the following two results as subroutines in our algorithm. Karloff [14] presents an NC^2 algorithm which calls a perfect matching algorithm (oracle) on $n+1$ graphs (each with $n-1$ or n vertices and at most m edges) in parallel, and which, assuming that these calls worked correctly, will announce *correctly* if G has a perfect matching; in any case, if it ever says that $|\mathrm{Mat}(G)| = 0$, then the number of perfect matchings is indeed zero. Actually, Karloff's result is more general; it works for maximum matchings. Kozen, Vazirani, and Vazirani [18] give a simple NC^2 algorithm to test, given a perfect matching, whether there are other perfect matchings in the graph. A direct extension of the technique yields an NC^2 procedure for checking, given a polynomial number of perfect matchings, whether there are other perfect matchings.

In conjunction with any Monte Carlo RNC^2 algorithm for finding a perfect matching (in particular, ours), Karloff's algorithm yields a Las Vegas RNC^2 algorithm for deciding if G has a perfect matching, and if so, for producing one. We first make the error probability of our perfect matching algorithm at most $\frac{1}{2(n+1)}$ (as mentioned in §2, this can be done in RNC^2) and, using only $O(n \log(m/n))$ random bits, then invoke Karloff's algorithm, which will make calls in parallel to our algorithm using the *same sequence of random bits* for each call. The probability that at least one of these calls produced a wrong result is at most $\frac{n+1}{2(n+1)} = 1/2$. Hence, we can decide if $|\mathrm{Mat}(G)| > 0$ *correctly*, using an expected $O(n \log(m/n))$ random bits. The above probability can be made any inverse polynomial with a processor penalty using the techniques of two-point sampling [5]. If we conclude that $|\mathrm{Mat}(G)| > 0$, then we run the algorithm of §3.2 to generate all the perfect matchings of G, assuming that $|\mathrm{Mat}(G)| \leq n^c + 1$. Finally, using the extension of the algorithm of Kozen, Vazirani, and Vazirani, we then check if there are other perfect matchings in the graph other than the ones generated, and accept if there are no other matchings; otherwise we report a failure. If G has at most n^c perfect matchings, then at most n^c perfect matchings will be generated and we will pass the test for other matchings, so the algorithm accepts correctly with high probability.

THEOREM 3.5. *There exists a Las Vegas RNC^2 algorithm for testing if a given graph G has $|\mathrm{Mat}(G)| \leq n^c$, whose expected randomness complexity is $O(n \log(m/n))$.*

3.3. Randomness-efficient algorithms with no information on $|\mathrm{Mat}(G)|$. The algorithms of Theorem 3.1 can be used to obtain randomness-efficient algorithms for perfect matching, even when no upper bound on the number of perfect matchings is given, by using the worst-case upper bound of $Z = (2m/n)^n$. However, the following algorithm gives a much better randomness complexity:

for $i := 0$ **to** $\lceil \log_2 \log_2((2m/n)^n) \rceil$ **do**
Let $Z = 2^{2^i}$ and run the algorithm of §3.1 to find a perfect matching;
exit if a perfect matching is found.

If there is at least one perfect matching, then this algorithm will find a perfect matching with good probability when $2^{2^i} \geq |\mathrm{Mat}(G)|$. If M is the number of perfect matchings then the number of random bits used and the time taken will be $O(\log M)$ and $O(\log^2 n \log\log M)$, respectively, with good probability, and so we get the following theorem.

THEOREM 3.6. *Let M=$|\mathrm{Mat}(G)|$. There is an RNC algorithm for finding a perfect matching in G with no given upper bound on M, which uses $O(\log M)$ random bits and runs in time $O(\log^2 n \log\log M) = O(\log^3 n)$ with high probability if $M > 0$; in the* worst case, *it uses $O(n \log(m/n))$ random bits and $O(\log^3 n)$ time.*

4. Other applications. The isolating lemma is a very abstract and powerful tool for making one solution from a possibly exponential-sized collection stand out. Because of its general nature, it has several applications. Thus, our generalized isolating lemma leads to randomness-efficient solutions to various problems, which we describe in this section.

4.1. Randomized reductions from SAT to USAT. An important result in complexity theory is the random reduction by Valiant and Vazirani from languages in *NP* to USAT, the language of uniquely satisfiable Boolean formulae [33]. This reduction is at the core of several fundamental results in complexity theory, most notably in the results of Toda on the unexpected power of counting classes [30]. The isolating lemma has been used to derive an alternative random reduction in [22]. We apply the isolating lemma to get a slightly different reduction and here; our generalization yields better randomness complexity.

Given a formula $F(x_1, x_2, \ldots, x_n)$ as input, to use the isolating lemma we let the ground set S be the set of variables $\{x_1, x_2, \ldots, x_n\}$ and let the family $\mathcal{F}$ be the satisfying assignments of F, where a satisfying assignment is represented by the subset of variables that are true in it. Assign weights $w_1, \ldots, w_n$ to the ground set elements as prescribed by the generalized isolating lemma and choose a random element y in the range $[1,n^8]$. Consider the following reduction.

On input $F(x_1, \ldots, x_n)$, $w_1, \ldots, w_n$, y we construct an *NP* machine M which works as follows: Guess an assignment for the variables $x_1, \ldots, x_n$. If the assignment satisfies the formula F and the weight of the assignment under the weight assignment $w_1, \ldots, w_n$ is equal to y then we accept. Note that since the weight assignment scheme succeeds with probability $1/2$, if F is satisfiable then with probability $1/2$ there is one minimum weight satisfying assignment. Since all the assignments have a weight in the range $[1, n^8]$ and y is chosen uniformly at random from $[1, n^8]$, we have

$$F \text{ is satsfiable} \Longrightarrow \text{Prob}[\, M \text{ accepts on exactly one path }] \geq \frac{1}{2 \times n^8}.$$

If F is not satisfiable then the machine M clearly accepts on no paths. Given this machine M, using the Cook–Levin reduction we can construct a boolean formula F' which has exactly the same number of satisfying assignments as the number of accepting paths of M. So the random reduction first assigns weights to the variables, chooses a random weight y, and then constructs the formula F' described above. By the description above it is easy to see that

$$F \text{ is satisfiable} \Longrightarrow \text{Prob}[\, F' \text{ is uniquely satisfiable }] \geq \frac{1}{2 \times n^8},$$

$$F \text{ is not satisfiable} \Longrightarrow \text{Prob}[\, F' \text{ is not satisfiable }] = 1.$$

This reduction requires $O(\log Z + \log n)$ random bits, where Z is an upper bound on the number of satisfying assignments of F. In the worst case this randomness complexity is $O(n)$, which improves on the $O(n \log n)$ bound of [22] and the $O(n^2)$ bound of [33] and matches the result of Tarui [27]. Thus we obtain the following theorem.

THEOREM 4.1. *There is a randomized reduction from SAT to USAT which uses* $O(\log Z + \log n)$ *random bits, where* n *and* Z *are the number of variables and an upper bound on the number of satisfying assignments, respectively.*

If the number of satisfying assignments of F is polynomially bounded as in case of languages in the class Few$P$, the reduction uses $O(\log n)$ random bits. This can be derandomized to yield new proofs of the results Few$P \subseteq \oplus P$ [3] and Few$P \subseteq C_{=}P$ [17] as described below.

DEFINITION. *A language* L *is in the class* Few$P$ *if there is a nondeterministic polynomial time machine* M *which accepts* L *and a polynomial* q *such that for all strings* x, $M(x)$ *accepts on at most* $q(|x|)$ *paths.*

Note that for a language L in Few$P$, given a string x, the Cook–Levin reduction produces a formula F with at most a polynomial number of satisfying assignments such that x is in L iff F is satisfiable.

DEFINITION. *A language* L *is in the class* $\oplus P$ *if there is a nondeterministic polynomial time machine* M *such that* x *is in* L *iff* M *on input* x *accepts on an odd number of paths.*

The class was first defined by Papadimitriou and Zachos [25], who also show that the class is closed under many operations; in fact, $P^{\oplus P} = \oplus P$. When we derandomize the above reduction for languages in Few$P$, we can get a polynomial number of formulas such that F is satisfiable iff one of these formulas is uniquely satisfiable. Also, if F is unsatisfiable then none of these formulas is satisfiable. Since the condition in the antecedents can easily be checked in $P^{\oplus P}$, we can get a new proof of the result that Few$P \subseteq \oplus P$.

DEFINITION. *A language* L *is in the class* $C_{=}P$ *iff there is a nondeterministic polynomial time machine* M *and a polynomial time computable function* f *such that* x *is in* L *iff* M *on input* x *accepts on exactly* $f(x)$ *paths.*

We can use $C_{=}P$ to test if one of a polynomial number of formulas has exactly one satisfying assignment as follows: Let $F_1, \ldots, F_t$ be formulas on n variables each with $s_1, \ldots, s_t$ satisfying assignments, respectively. If we could construct a formula F' with $(s_1 - 1)\ldots(s_t - 1)$ satisfying assignments, then F' would have exactly zero satisfying assignments iff one of the F_j's has exactly one satisfying assignment. However, doing this directly involves constructing formulas with one less satisfying assignment than the F_j's, which is highly intractable [24]. We do this slightly indirectly as follows. Each term in the product $(s_1 - 1)\ldots(s_t - 1)$ is of the form $\pm s_{i_1} s_{i_2} \ldots s_{i_r}$, where $r \leq t$. We can easily construct a formula which has $s_{i_1} s_{i_2} \ldots s_{i_r}$ satisfying assignments; so if the coefficient of this term is $+1$ then we are done. However, if the coefficient is -1 then we construct a formula with $2^{p(n)} - s_{i_1} s_{i_2} \ldots s_{i_r}$ satisfying assignments, where $p(n) = t \times n$. A nondeterministic polynomial time machine M which accomplishes this works as follows. First, M nondeterministically chooses a term in the product $(s_1 - 1)\ldots(s_t - 1)$. If the term is of the form $s_{i_1} s_{i_2} \ldots s_{i_r}$, it guesses r assignments and accepts iff the assignments satisfy $F_{i_1}, \ldots, F_{i_r}$, respectively. If the term is of the form $-1 \times s_{i_1} s_{i_2} \ldots s_{i_r}$, it guesses t assignments and accepts iff the first r formulas do not satisfy $F_{i_1}, \ldots, F_{i_r}$ or the last $t - r$ assignments are all zeros. It can be easily verified that if the term has coefficient 1 then M accepts on $s_{i_1} s_{i_2} \ldots s_{i_r}$ paths, and if the coefficient is -1 then M accepts on $2^{p(n)} - s_{i_1} s_{i_2} \ldots s_{i_r}$ paths. Since the number of terms with coefficient -1 is 2^{t-1}, M accepts on $(s_1 - 1)\ldots(s_t - 1) + 2^{p(n)} \times 2^{t-1}$ paths. Using the Cook–Levin reduction, one can construct a formula G with exactly the same number of satisfying assignments. Now G has exactly $2^{p(n)} \times 2^{t-1}$ satisfying assignment iff one of the original formulas has exactly one satisfying assignment. Thus the reduction also gives a new proof of Few$P \subseteq C_{=}P$.

Here again we wish to emphasize that although there are many proofs of the above two results, our proofs follow directly from bounding the randomness complexity of the general randomized reduction.

4.2. Improved parallel randomness complexity for problems on matroids. The isolating lemma has been used to obtain *RNC* algorithms for basic problems on linearly representable matroids such as matroid intersection and matching (Narayanan, Saran, and Vazirani [23]). These problems are generalizations of bipartite and general graph matching.

DEFINITION. *A set system $M = (S, \mathcal{F})$ is a* matroid *if the following hold:*

- $\phi \in \mathcal{F}$,
- *if* $A \subseteq B \subseteq S$ *and* $B \in \mathcal{F}$, *then* $A \in \mathcal{F}$, *and*
- *if* $A_1 \in \mathcal{F}$ *and* $A_2 \in \mathcal{F}$ *with* $|A_2| = |A_1| + 1$, *then there exists* $x \in A_2 - A_1$ *such that* $A_1 \cup \{x\} \in \mathcal{F}$.

Every element of $\mathcal{F}$ is called an *independent set*. A basic consequence of the matroid axioms is that every maximal independent set is also a maximum independent set, and the cardinality of any maximum independent set of M is also called the *rank* of M. A matroid M of rank r over a ground set S of cardinality n is *linearly representable* over a field $\mathbf{F}$ if there exists a matrix $C \in \mathbf{F}^{r \times n}$ whose columns are indexed by the elements of S, such that a subset of S is independent in M iff the corresponding set of columns of C are linearly independent over $\mathbf{F}$. M is *linearly represented* if it is presented as the matrix B. Henceforth, the field $\mathbf{F}$ is assumed to be the field of rationals, $\mathbf{Q}$. All the results of [23] and this section hold *only* for linearly represented matroids. For a good introduction to the theory of matroids see Welsh [35]. For other work on randomized algorithms for the matroid problem see Camerini, Galbiati, and Maffioli [4].

The matroid intersection problem is to find a maximum cardinality independent set in both of two given matroids M_1 and M_2, each of rank r and over the same ground set S of cardinality n. In this case, we observe that the size of the family on which the isolating lemma operates, as presented in [23], is at most $I \times \left(\binom{c}{r-h}\right)^2 \times (r-h)!$, where h is the size of the largest set in $M_1 \cap M_2$ and I is the number of sets of size h in $M_1 \cap M_2$. Since this can be bounded by $(nr^2)^h \times r! \leq (nr^2)^r \times r^r \leq n^{4r}$, we can use generalized isolation to reduce the randomness complexity from the previous bound of $O((n + r^2)\log n)$ [23] to $O(r \log n)$, improving by a factor of at least $\Omega(\sqrt{n})$ in all cases and by more in general. If $h \geq r - O(\log_r n)$ and there is a polynomial (in n) bound on I then, since $I\left(\binom{r}{r-h}\right)^2 \times (r-h)! = n^{O(1)}$, we get the first *NC* algorithms. In the case of bipartite matching, which is a special case of matroid intersection, $h = r$ and a polynomial bound on I corresponds exactly to a polynomial bound on the number of matchings in the input graph. Thus, this gives one way to generalize the results of [11] to matroids.

THEOREM 4.2. *Matroid intersection for linearly represented matroids of rank r can be solved in RNC^2 using $O(r \log n)$ random bits. If the cardinality of the maximum intersection is given to be at least $r - O(\log_r n)$, and if the number of maximum cardinality intersections is given to be bounded by a given polynomial of n, then it can be solved in NC^2.*

For linearly represented vectors, the matroid matching problem is as follows: Given m pairs of vectors over $\mathbf{Q}^{2n}$, pick the largest number of pairs such that the vectors picked are linearly independent. For this problem the RNC^2 algorithm of [23, Thm. 4.3] uses the isolating lemma on a set family over a ground set of $m + \binom{2n}{2}$ elements and at most

$$\binom{m + \binom{2n}{2}}{n} \leq (m + 2n^2)^n$$

subsets, and thus, by invoking the generalized isolating lemma, we improve the randomness complexity from $O((m + n^2)\log(m + n^2))$ [23] to $O(n \log(m + n^2))$. We can count the size

of the family to which the isolating lemma is applied more carefully to bound the randomness complexity of the algorithm to be $O(\log(I \times \binom{2n}{2n-2h} \times (2n-2h)!))$, where h is the maximum number of pairs that can be picked so that the union is linearly independent and I is the number of ways these h pairs can be picked. As in the case of matroid intersection we can obtain deterministic algorithms if I is polynomial and if $h \geq n - \log_n m$. Tiwari, using sparse interpolation techniques, gives *NC* algorithms for this problem when the number of full dimensional solutions (i.e., when $h = n$) is given as polynomially bounded in n and m [29]. Thus, our result yields a generalization.

THEOREM 4.3. *Matroid matching can be solved in RNC*^{2} *using* $O(n \log(m+n^2))$ *random bits. If* $h \geq n - \log_n m$ *is the size of the largest number of pairs whose union is linearly independent, and there is a polynomial bound on the number of ways such a set of h pairs can be picked, matroid matching can be solved in NC*^{2}.

5. Conclusions and open problems. The main contribution of this work is our generalized isolating lemma. For various problems solvable via randomness in general [22], [23], [33] and deterministically when the number of solutions is small [18], [11], [3], this lemma bounds their randomness complexity in terms of the number of solutions. Our results span the spectrum of the number of solutions, and imply and sometimes improve the previously known results at the extremes. On the other hand, our lower bound for isolation implies that attempts to solve these problems deterministically based on isolation must also exploit their structure, rather than view them merely as abstract unknown collections of sets. An obvious interesting question that arises here is as follows: Can we use some knowledge of the structure of the solution space, from which we are trying to isolate one solution, to reduce the randomness complexity of isolation or even perhaps obtain deterministic algorithms? In case of the perfect matching problem it would be interesting to see if we can obtain deterministic algorithms for special classes of graphs with this framework.

Another direct open problem falling out of the generalized isolating lemma is the reduction of the magnitude of the weights assigned by it to the elements of the ground set; currently, they can be as high as N^7. In the context of matching, this will lead to more processor-efficient *RNC* algorithms. For the random reduction a decrease in the magnitude of the weights of the elements will increase the success probability of the reduction.

Acknowledgments. We thank our advisors Juris Hartmanis and David Shmoys for their guidance, suggestions, and support. This paper has also benefited greatly from discussions with several other people. We are particularly indebted to Joseph Cheriyan, Steve Mitchell, Moni Naor, Éva Tardos, and Vijay Vazirani. We would like to thank Martin Tompa for making valuable comments on the conference proceedings version of this paper.

REFERENCES

[1] L. ADLEMAN, *Two theorems on random polynomial time*, in Proc. IEEE Symposium on Foundations of Computer Science, 1978, pp. 75–83.

[2] A. BORODIN, S. A. COOK, AND N. PIPPENGER, *Parallel computation for well-endowed rings and space-bounded probabilistic machines*, Inform. Control, 58 (1983), pp. 113–119.

[3] J.-Y. CAI AND L. HEMACHANDRA, *On the power of parity polynomial time*, Math. Systems Theory, 23 (1990), pp. 95–106.

[4] P. M. CAMERINI, G. GALBIATI, AND F. MAFFIOLI, *Random pseudopolynomial algorithms for exact matroid problems*, J. Algorithms, 13 (1992), pp. 258–273.

[5] B. CHOR AND O. GOLDREICH, *On the power of two-point sampling*, J. Complexity, 5 (1989), pp. 96–106.

[6] L. CSANKY, *Fast parallel matrix inversion algorithms*, SIAM J. Comput., 5 (1976), pp. 618–623.

[7] E. DAHLHAUS AND M. KARPINSKI, *The matching problem for strongly chordal graphs is in NC*, Tech. report 855-CS, University of Bonn, 1986.

[8] N. DE BRUIJN, *On the number of positive integers* $\leq x$ *and free of prime factors* $> y$, Proc. Kon. Nederl. Akad. Wetensch, A54 (1951), pp. 50–60.

[9] A. V. GOLDBERG, S. A. PLOTKIN, D. B. SHMOYS, AND É. TARDOS, *Using interior-point methods for fast parallel algorithms for bipartite matching and related problems*, SIAM J. Comput., 21 (1992), pp. 140–150.

[10] A. V. GOLDBERG, S. A. PLOTKIN, AND P. M. VAIDYA, *Sublinear-time parallel algorithms for matching and related problems*, J. Algorithms, 14 (1993), pp. 180–213.

[11] D. Y. GRIGORIEV AND M. KARPINSKI, *The matching problem for bipartite graphs with polynomially bounded permanents is in NC*, in Proc. IEEE Symposium on Foundations of Computer Science, 1987, pp. 166–172.

[12] D. Y. GRIGORIEV, M. KARPINSKI, AND M. F. SINGER, *Fast parallel algorithms for multivariate polynomial interpolation over finite fields*, SIAM J. Comput., 19 (1990), pp. 1059–1063.

[13] L. K. GROVER, *Fast parallel algorithms for bipartite matching*, Proc. of the Integer Programming and Combinatorial Optimization Conference, 1992, pp. 367–384.

[14] H. J. KARLOFF, *Las Vegas RNC algorithm for maximum matching*, Combinatorica, 6 (1986), pp. 387–391.

[15] R. M. KARP, E. UPFAL, AND A. WIGDERSON, *Constructing a perfect matching is in random NC*, Combinatorica, 6 (1986), pp. 35–48.

[16] P. W. KASTELYN, *Graph theory and crystal physics*, in Graph Theory and Theoretical Physics, F. Harary, ed., Academic Press, New York, 1967, pp. 43–110.

[17] J. KÖBLER, U. SCHÖNING, S. TODA, AND J. TORAN, *Turing machines with few accepting computations and low sets for PP*, J. Comput. System Sci., 44 (1992), pp. 272–286.

[18] D. KOZEN, U. V. VAZIRANI, AND V. V. VAZIRANI, *NC algorithms for comparability graphs, interval graphs, and testing for unique perfect matching*, in Proc. FST and TCS Conference, Lecture Notes in Computer Science 206, Springer-Verlag, Berlin, 1985, pp. 496–503.

[19] C. H. C. LITTLE, *An extension of Kastelyn's method of enumerating the 1-factors of planar graphs*, in Combinatorial Mathematics, Proc. Second Australian Conference, Lecture Notes in Mathematics 403, Springer-Verlag, Berlin, 1974, pp. 63–72.

[20] L. LOVÁSZ AND M. PLUMMER, *Matching Theory*, North–Holland, Amsterdam, 1986.

[21] G. L. MILLER AND J. NAOR, *Flow in planar graphs with multiple sources and sinks*, in Proc. IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1989, pp. 112–117.

[22] K. MULMULEY, U. V. VAZIRANI, AND V. V. VAZIRANI, *Matching is as easy as matrix inversion*, Combinatorica, 7 (1987), pp. 105–113.

[23] H. NARAYANAN, H. SARAN, AND V. V. VAZIRANI, *Randomized parallel algorithms for matroid union and intersection, with applications to arborescences and edge-disjoint spanning trees*, in Proc. ACM–SIAM Symposium on Discrete Algorithms, 1992, pp. 357–366.

[24] M. OGIWARA AND L. HEMACHANDRA, *Complexity theory for feasible closure properties*, in Proc. of the Sixth Annual Conference on Structure in Complexity Theory, 1991, pp. 16–27.

[25] C. PAPADIMITRIOU AND S. ZACHOS, *Remarks on the power of counting*, in Proc. 6th GI Conference on Theoretical Computer Science, Lecture Notes in Computer Science 145, Springer-Verlag, 1983, pp. 276–296.

[26] C. H. PAPADIMITRIOU AND M. YANNAKAKIS, *The complexity of restricted spanning tree problems*, J. Assoc. Comput. Mach., 29 (1982), pp. 285–309.

[27] J. TARUI, *Randomized polynomials, threshold circuits and the polynomial hierarchy*, in Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science 480, 1991, pp. 238–250.

[28] W. THRASH, *A note on the least common multiples of dense sets of integers*, Tech. report 93-02-04, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1993.

[29] P. TIWARI, *Parallel algorithms for instances of linear matroid parity with small number of solutions*, Tech. report RC 12766, IBM T. J.Watson Research Center, 1987.

[30] S. TODA, *PP is as hard as the polynomial-time hierarchy*, SIAM J. Comput., 20 (1991), pp. 865–877.

[31] M. TOMPA, *Lecture notes on probabilistic algorithms and pseudorandom generators*, Tech. report 91-07-05, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1991.

[32] P. M. VAIDYA, *Reducing the parallel complexity of certain linear programming problems*, in Proc. IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1990, pp. 583–589.

[33] L. G. VALIANT AND V. V. VAZIRANI, *NP is as easy as detecting unique solutions*, Theoret. Comput. Sci., 47 (1986), pp. 85–93.

[34] V. V. VAZIRANI, *NC algorithms for computing the number of perfect matchings in $K_{3,3}$-free graphs and related problems*, Inform. Comput., 80 (1989), pp. 152–164.

[35] D. J. A. WELSH, *Matroid Theory*, Academic Press, New York, 1976.

SIAM J. COMPUT.
Vol. 24, No. 5, pp. 1051–1067, October 1995

007

MINIMUM COST PATHS IN PERIODIC GRAPHS*

FRANZ HÖFTING† AND EGON WANKE‡

Abstract. We consider graphs with d-dimensional integral vector weights and rational cost values associated with the edges. We analyze the problem of finding a minimum cost path between two given vertices such that the vector sum of all edges in the path equals a given target vector m. The present paper shows that there are polynomial time algorithms for finding such a minimum cost m-path if the dimension of the vector weights is bounded by a constant and the vector weights are represented unary, where the general version is NP-complete under various restrictions.

Key words. periodic graphs, shortest paths, NP-completeness, integer linear programming, scheduling

AMS subject classifications. 05C12, 05C38, 68Q25, 68R10

1. Introduction and summary.

A d-dimensional *periodic graph* G^∞ is an infinite graph finitely described by a so-called *static graph* G. A static graph is a directed graph with d-dimensional integral vector weights associated with the edges. The periodic graph G^∞ is obtained by placing the static graph G in a d-dimensional orthogonal grid. That is, a copy of the vertex set of G is placed at each point in the integral lattice $\mathbb{Z}^d$. Then, for each edge of G from u to v with vector weight t, the copy of u at each lattice point z is connected with the copy of v at lattice point $z+t$. The connection pattern between the vertices is the same everywhere in the grid. A periodic graph can be regarded as *undirected* by ignoring the direction of its edges. For our purposes, this is equivalent to the property of the static graph that, for each edge from u to v with vector weight t, there is an edge from v to u with vector weight $-t$.

The static/periodic graph model has been considered by several authors with respect to various applications. Karp, Miller, and Winograd [8] analyze systems of *uniform recurrence equations*, where the variables are indexed by integral vectors. The dependencies of the variables in the equations are modeled by a so-called infinite *dependence graph*, which corresponds to a periodic graph. The complete dependence graph can be represented by a so-called *reduced dependence graph*, which corresponds to a static graph.

There are several authors who also consider the static/periodic graph model. Rao [15] considers *regular iterative algorithms* and their implementation on processor arrays. He introduces projections that map certain finite parts of periodic graphs onto finite processor arrays for possible systolic computations. Iwano and Steiglitz [6], Kosaraju and Sullivan [10], and Cohen and Megiddo [3] cite applications of static/periodic graphs in modeling very large scale integration (VLSI) circuits. They consider the problem of finding cycles in directed periodic graphs or, equivalently, of finding so-called *zero cycles* in static graphs. Cohen and Megiddo also show that the zero cycle problem can be solved in strongly polynomial time if the dimension of the vector weights is bounded by a constant. They also show that the general problem is equivalent to linear programming. In [13] Orlin provides some polynomial time algorithms for graph problems on one-dimensional periodic graphs. In [9] Kodialam and Orlin develop an efficient algorithm to determine the strongly connected components of a periodic graph. Cohen and Megiddo [4] develop a linear time algorithm for testing planarity of periodic graphs. They also develop algorithms for bipartiteness, connectivity, and minimum cost spanning forests. Backes, Schwiegelshohn, and Thiele [1] analyze the structure of longest paths in periodic graphs represented by strongly connected static graphs.

*Received by the editors July 15, 1992; accepted for publication (in revised form) April 22, 1994.

†Department of Computer Science, University of Paderborn, D-33098 Paderborn, Germany (fh@uni-paderborn.de).

‡Department of Computer Science, Mathematical Institute, Heinrich-Heine-Universität, Düsseldorf, D-40225 Düsseldorf, Germany (wanke@cs.uni-duesseldorf.de).

They show that strongly connected static graphs represent periodic graphs in that almost all longest paths follow some regular structure if the path lengths exceed a certain threshold.

In this paper, we consider the problem of finding a minimum cost path between two given vertices in a static graph such that the vector sum of all edges in the path equals a given target vector m. This problem is called the MINIMUM COST m-PATH problem. All of our results also hold for maximum cost m-paths. When modeling dependencies of variables, as, for example, in uniform recurrence equations, where edges represent data dependencies and the cost values of the edges correspond to computation times, the values of certain maximum cost paths correspond to lower bounds on an optimal schedule for the computation.

Our main results can be summarized as follows. In §3 we establish a close relationship between the MINIMUM COST m-PATH problem and integer linear programming by giving a formulation in terms of an integer linear program. From this formulation we derive the containment of MINIMUM COST m-PATH in NP. It is also shown that even very restricted versions of the problem remain NP-hard. The m-PATH problem for undirected periodic graphs is shown to be in P.

In §4, in the case where the dimension of the vector weights of the edges is a constant and the vector weights are given unary, it is shown how it is possible to decide in polynomial time whether there is a minimum cost m-path, and if so, how to determine one. Note that m is not required to be given unary.

In §5, we present a very general upper bound for the maximal number of edges in a minimum cost m-path. This bound can be used to control shortest path algorithms to find a minimum cost m-path, if one exists. Such algorithms are interesting from a practical point of view.

2. The problem. A *static graph* $G = (V, E)$ is a directed graph with a vertex set V and some edge set

$$E \subseteq V \times V \times \mathbb{Z}^d \times \mathbb{Q}.$$

Each edge $e = (u, v, t, c)$ has a *source vertex* $\text{source}(e) = u$, a *target vertex* $\text{target}(e) = v$, a *transit vector* $\text{tran}(e) = t$, and a *cost value* $\text{cost}(e) = c$. This definition allows multiple edges, i.e., several edges with the same source and target vertex.

The *periodic graph* $G^\infty = (V^\infty, E^\infty)$ associated with a d-dimensional static graph $G = (V, E)$ is the infinite directed graph defined by the vertex set

$$V^\infty = V \times \mathbb{Z}^d$$

and the edge set

$$E^\infty = \{((u, x), (v, x + t), c) | (u, v, t, c) \in E\} \subseteq V^\infty \times V^\infty \times \mathbb{Q}.$$

G^∞ is considered to be *undirected* if, for each edge $((u, x), (v, y), c)$, there is an edge $((v, y), (u, x), c)$; or, equivalently, for each edge (u, v, t, c) in G there is an edge $(v, u, -t, c)$ in G. Figures 2.1 and 2.2 show a static graph G and a certain part of the infinite periodic graph G^∞, respectively.

Let G be a d-dimensional static graph. A *path* p from a vertex u to a vertex v is an alternating sequence

$$(u_1, e_1, u_2, \ldots, u_n, e_n, u_{n+1})$$

of vertices and edges such that $u = u_1$, $v = u_{n+1}$, $\text{source}(e_i) = u_i$, $\text{target}(e_i) = u_{i+1}$ for $i = 1, \ldots, n$. Path p is a *cycle* if $u = v$ and is called *simple* if all vertices (except the first and

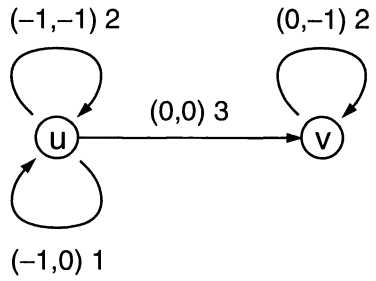


FIG. 2.1. *A two-dimensional static graph G. Edges are marked with two-dimensional transit vectors and cost values.*

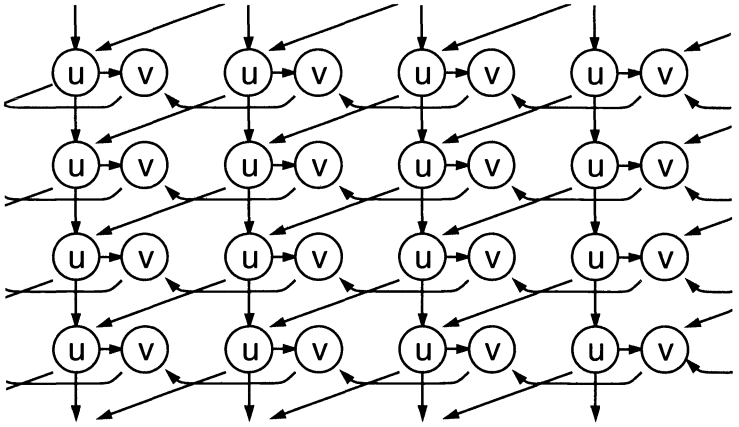


FIG. 2.2. *A part of the infinite periodic graph defined by the static graph from Fig. 1. For the sake of readability, the cost values are omitted.*

the last vertex) are pairwise distinct. The transit vector and cost value of p are defined by

$$\text{tran}(p) := \sum_{i=1}^{n} \text{tran}(e_i)$$

and

$$\text{cost}(p) := \sum_{i=1}^{n} \text{cost}(e_i),$$

respectively. A path (cycle) p is also called a tran(p)-path (-cycle).

A path p from u to v is a *minimum cost path* if there is no other path q from u to v in G such that cost(q) < cost(p). There are two cases in which, for some m, a minimum cost m-path in a static graph does not need to exist. First, the target vertex need not be reachable from the source vertex by a path with transit vector m, (see Example 2.1); second, for each rational b there is an m-path from the source vertex to the target vertex, but with less cost than b (see Example 2.1).

Example 2.1. Figure 2.3 shows a static graph G with the following properties:

- G has a path from u to v but, for example, no (2, 6)-path from u to v.
- For each rational b there is a (2, 5)-path from u to v with cost less than b.
- G^{∞} has no cycle.

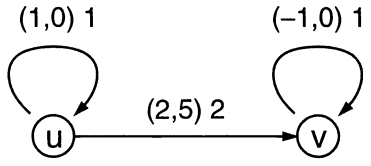


FIG. 2.3. *A static graph G.*

To state our problem properly, we define min_cost(G, u, v, m) as the cost of a minimum cost m-path from u to v in a static graph G, where

$$\text{min\_cost}(G, u, v, m) = \text{undef}$$

if there is no m-path from u to v in G, and

$$\text{min\_cost}(G, u, v, m) = -\infty$$

if, for each rational b, there is an m-path from u to v in G with cost less than b.

For an integer a, we denote by $\vec{a}$ the integral vector with entry a at each coordinate. The dimension of $\vec{a}$ will either be unique or unimportant at the moment in which $\vec{a}$ will be used. It will also be unique from the context of whether $\vec{a}$ is a row or column vector.

In this paper we investigate the following problem.

MINIMUM COST m-PATH
Instance: A d-dimensional static graph $G = (V, E)$, two vertices u and v of G, and a target vector $m \in \mathbb{Z}^d$.
Problem: Compute min_cost(G, u, v, m).

Each path from (u, x) to $(v, x + m)$ in G^∞, for any $x \in \mathbb{Z}^d$, corresponds to an m-path from u to v in G. Since G^∞ is infinite, the case in which min_cost$(G, u, v, m) = -\infty$ does not necessarily imply that G^∞ has a negative cost cycle. G^∞ may also contain an infinite number of negative cost paths from u to v with transit vector m, as shown in Example 2.1.

We refer to the above problem as the m-PATH problem if we neglect the cost values of the edges and simply ask for existence of an m-path starting at u and ending at v.

3. The general solution. In this section, we will show that min_cost(G, u, v, m) can be computed with methods from *integer linear programming.*[1] Let $G = (V, E)$, $V = \{u_1, \ldots, u_n\}$, $E = \{e_1, \ldots, e_k\}$, $m = (m_1, \ldots, m_d)$, and $u, v \in V$.

We define

- a row vector $c = (\text{cost}(e_1), \ldots, \text{cost}(e_k))$;
- an $(n + d) \times k$ matrix A with

$$A(i, j) = \begin{cases} -1 & \text{if } 1 \le i \le n \text{ and } u_i = \text{source}(e_j), \\ 1 & \text{if } 1 \le i \le n \text{ and } u_i = \text{target}(e_j), \\ \text{tran}(e_j)_{i-n} & \text{if } n < i \le n + d, \\ 0 & \text{otherwise;} \end{cases}$$

- a column vector $b = (b_1, \ldots, b_{n+d})^T$ with

$$b_i = \begin{cases} -1 & \text{if } 1 \le i \le n \text{ and } u_i = v, \\ 1 & \text{if } 1 \le i \le n \text{ and } u_i = w, \\ m_{i-n} & \text{if } n < i \le n + d, \\ 0 & \text{otherwise.} \end{cases}$$

[1] For background information about integer linear programming see, for example, [12] and [16].

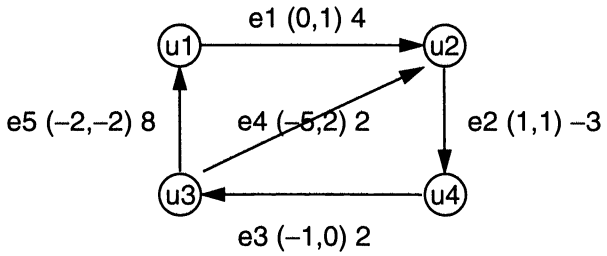


FIG. 3.1. *A two-dimensional static graph. Edges are marked by names, two-dimensional integral vectors, and cost values.*

Example 3.1. For the static graph G from Fig. 3.1, source vertex u_1, target vertex u_3, and $m = (-5, 5)$, we have

$$c = (\ 4,\ \ -3,\ \ 2,\ \ 2,\ \ 8\ ),$$

$$A = \begin{pmatrix} -1 & 0 & 0 & 0 & 1 \\ 1 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & -1 & -1 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 1 & -1 & -5 & -2 \\ 1 & 1 & 0 & 2 & -2 \end{pmatrix}, \quad \text{and } b = \begin{pmatrix} -1 \\ 0 \\ 1 \\ 0 \\ -5 \\ 5 \end{pmatrix},$$

where $V = \{u_1, u_2, u_3, u_4\}$ and $E = \{e_1, e_2, e_3, e_4, e_5\}$.

A graph G is called *connected*[2] if, for each pair of vertices u, v of G, the edges in G can be redirected such that there is a simple path from u to v in G. An *edge set* $F \subseteq E$ of G is called *connected* if the graph $H = (U, F)$ with vertex set

$$U = \{u | u \text{ is an end vertex of an edge from } F\}$$

is connected.

A careful investigation shows that the cost of a minimum cost m-path from u to v in G is exactly

$$\min \left\{ c \cdot x \;\middle|\; \begin{array}{l} A \cdot x = b, \\ x \text{ is a nonzero and nonnegative integral column vector, and} \\ \{e_j \in E | x_j > 0\} \text{ is a connected edge set} \end{array} \right\}.$$

Assume that $A \cdot x = b$, where x is a nonnegative integral column vector. The positive entries x_j in x can be considered as the frequencies of the edges e_j in a possible m-path from u to v. Then, the -1 in the upper part of b ensures that vertex u is left once more than it is reached, the 1 in the upper part of b ensures that vertex v is reached once more than it is left, and the zero entries in the upper part of b ensure that all other vertices are reached as often as they are left. Because of Euler and one of his first results in graph theory, it follows that the edges e_j with $x_j > 0$ represent a path if the graph defined by these edges and their end vertices is connected. Finally, the target vector m in the lower part of b ensures that the path is an m-path.

The formulation of the MINIMUM m-PATH problem above yields the proof of the following theorems.

[2] A connected directed graph is also called weakly connected.

THEOREM 3.2. *The MINIMUM COST m-PATH problem belongs to NP.*

Proof. Choose a connected edge set $F \subseteq E$ and solve the system of integer linear equations (integer linear programming belongs to NP [2])

$$\min \left\{ c \cdot x \;\middle|\; \begin{array}{l} A \cdot x = b, \\ x \text{ is a positive integral column vector} \end{array} \right\},$$

where c, A, and b are defined as above, but now, for the *connected* graph, $H = (U, F)$ with vertex set

$$U = \{u | u \text{ is an end vertex of an edge from } F\}. \qquad \square$$

THEOREM 3.3. *m-PATH is NP-hard even if*

1. *the static graph G has exactly one vertex and*
   - $m = \vec{1}$ *and all entries in the transit vectors are from* $\{0, 1\}$, *or*
   - *m is one-dimensional and all transit integers are positive, or*
2. *G has at most two vertices and*
   - $m = \vec{0}$ *and all entries in the transit vectors are from* $\{-1, 0, 1\}$, *or*
   - *m is one-dimensional and zero and all transit integers are positive.*

Proof. The question of whether a system

$$A \cdot x = m^T$$

has a nonnegative integral solution is NP-hard even if all components in A are from $\{0, 1\}$ and $m = \vec{1}$, or if A is a positive row vector (see [7]). The first problem is called ZERO-ONE INTEGER PROGRAMMING, the second is called SUBSET SUM. This proves the first part of the theorem.

For the second part, let G be the corresponding single-vertex static graph G defined by an instance of ZERO-ONE INTEGER PROGRAMMING and SUBSET SUM, respectively. Let u be the single vertex in G. Insert a new vertex v to G and an edge from u to v with transit vector $-m$. Then there is an m-path in G from u to u if and only if there is a $\vec{0}$-path from u to v in the extended graph G. $\square$

THEOREM 3.4. *The m-PATH problem is solvable in polynomial time if the static graph defines an undirected periodic graph.*

Proof. Without loss of generality, we can assume that G is connected, otherwise we consider only the connected component containing u and v. It is easy to see that there is an m-path from u to v in G if and only if $A \cdot x = b$ has an integral solution in which x is not restricted to be nonnegative. Since G^∞ is considered to be undirected, we can assume that for each edge e in G there is an edge e' in G such that source(e) = target(e'), target(e) = source(e'), and tran$(e) = -$tran(e'). That is, if $A \cdot x = b$ then $A \cdot (x_1 + y, \ldots, x_k + y) = b$ for each integer y. Thus $A \cdot x = b$ also has a positive integral solution if it has an integral solution. Such systems of *linear diophantine equations* are solvable in polynomial time; see, for example, [16]. $\square$

THEOREM 3.5. *The MINIMUM COST m-PATH problem for static graphs which define undirected periodic graphs is NP-hard even if*

1. *the static graph G has exactly one vertex and*
   - $m = \vec{1}$, *all entries in the transit vectors of the edges are from* $\{0, 1\}$, *and all entries in the cost values are* 1, *or*
   - *m is one-dimensional and all entries in the transit vectors and cost values of the edges are nonnegative integers, or*
2. *the static graph G has at most two vertices and*
   - $m = \vec{0}$, *all entries in the transit vectors of the edges are from* $\{-1, 0, 1\}$, *and all entries in the cost values are* 1, *or*

- *m is one-dimensional and all entries in the transit vectors are integers, and all entries in the cost values are* 1.

Proof. Finding minimal solutions for ZERO-ONE linear diophantine equations or one general linear diophantine equation is NP-complete; see [16]. The second part can be shown by an extension similar to the one in the proof of Theorem 3.3. □

4. Pseudopolynomial time solutions. Now we present a pseudopolynomial time algorithm for MINIMUM COST m-PATH. This algorithm takes polynomial time if the dimension of the static graphs is constant and the integers in the transit vectors are given unary. If the static graphs have only one vertex, then there is a pseudopolynomial time solution in accordance with the results of Papadimitriou [14]. In [14] it is shown that there is a pseudopolynomial time algorithm for integer linear programming if the number of constraints is fixed.

To illustrate the restrictions we impose on the instances of MINIMUM COST m-PATH, we first carefully define the size of an instance. The *size* of an integer k, denoted by size(k), is the size of its binary representation. The *size* of an integral vector $t = (t_1, \ldots, t_k)$, denoted by size(t), is the sum of all size(t_i) for $i = 1, \ldots, k$. The *size* of a static graph G, denoted by size(G), is its number of vertices and edges plus the sum of all size(tran(e)) for all edges e in G. The sizes of the rational cost values associated with the edges are not explicitly involved in our complexity analysis and are assumed to be constant.

For a vector $t = (t_1, \ldots, t_d)$ or a set $t = \{t_1, \ldots, t_d\}$ of integers, let $\|t\|$ be the maximal absolute entry of all t_i's, i.e.,

$$\|t\| = \max_{i=1,\ldots,d} |t_i|.$$

The following fact is easy to verify and may be easy to improve.

FACT 4.1. *Let $G = (V, E)$ be a static graph and*

$$t_{\max} := \max\{\|tran(e)\| \mid e \in E\}.$$

1. *Let p be a path in G with exactly n edges; then $\|\text{tran}(p)\| \leq t_{\max} \cdot n$.*
2. *All paths in G with at most n edges have at most $(2 \cdot t_{\max} \cdot n + 1)^d$ different transit vectors.*
3. *Given two vertices u and v and a transit vector m, the cost of a minimum cost m-path from u to v in G with at most n edges is computable in time*

$$n \cdot |V| \cdot |E| \cdot (2 \cdot t_{\max} \cdot n + 1)^d.$$

This can be shown by a breadth-first-search processing with depth at most n and breadth at most $|V| \cdot (2 \cdot t_{\max} \cdot n + 1)^d$ at each level.

4. *Given two vertices u and v, a target vector m, and a set U of at most k vertices, the cost of a minimum cost m-path from u to v in G with at most n edges which passes all vertices from U is computable in time*

$$2^k \cdot n \cdot |V| \cdot |E| \cdot (2 \cdot t_{\max} \cdot n + 1)^d.$$

This can be shown by a breadth-first-search processing with depth at most n and breadth at most $2^k \cdot |V| \cdot (2 \cdot t_{\max} \cdot n + 1)^d$ at each level.

In the subsequent proofs, we frequently use the following path-decomposition technique: Suppose

$$p = (u_1, e_1, u_2, \ldots, u_{n-1}, e_{n-1}, u_n)$$

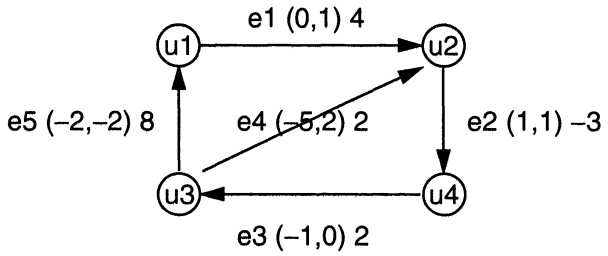


FIG. 4.1. *The two-dimensional static graph analyzed in Example* 4.2.

is a path in a graph G. Let

$$q = (u_i, e_i, u_{i+1}, \ldots, u_{i+l}, e_{i+l}, u_{i+l+1})$$

be a simple cycle of length l contained in p such that all inner vertices $u_{i+1}, \ldots, u_{i+l}$ of q are contained at least once somewhere else in p. (Since q is a cycle, the vertices u_i and u_{i+l+1} are the same vertices.) Now cut out the simple cycle q from the path p. It remains a path

$$p' = (u_1, e_1, u_2, \ldots, u_i, e_{i+l+1}, u_{i+l+2}, \ldots, u_{n-1}, e_{n-1}, u_n)$$

in G with l vertices and l edges less than before. Repeat this cycle elimination until such cycles do not exist anywhere in the remaining path. Then each vertex from the initial path p is contained at least once in the remaining path p' (see Example 4.2).

Example 4.2. Consider the $(-7, 5)$-path

$$p = (u_1, e_1, u_2, e_2, u_4, e_3, u_3, e_5, u_1, e_1, u_2, e_2, u_4, e_3, u_3, e_4, u_2, e_2, u_4, e_3, u_3)$$

with cost 15 in the graph from Fig. 4.1. This path p can be decomposed into a simple path

$$p' = (u_1, e_1, u_2, e_2, u_4, e_3, u_3)$$

with cost 3 and two simple cycles

$(u_1, e_1, u_2, e_2, u_4, e_3, u_3, e_5, u_1)$ and $(u_2, e_2, u_4, e_3, u_3, e_4, u_2)$, or
$(u_1, e_1, u_2, e_2, u_4, e_3, u_3, e_5, u_1)$ and $(u_4, e_3, u_3, e_4, u_2, e_2, u_4)$, or
$(u_1, e_1, u_2, e_2, u_4, e_3, u_3, e_5, u_1)$ and $(u_3, e_4, u_2, e_2, u_4, e_3, u_3)$, or
$(u_2, e_2, u_4, e_3, u_3, e_5, u_1, e_1, u_2)$ and $(u_2, e_2, u_4, e_3, u_3, e_4, u_2)$, or
$(u_2, e_2, u_4, e_3, u_3, e_5, u_1, e_1, u_2)$ and $(u_4, e_3, u_3, e_4, u_2, e_2, u_4)$, or
$(u_2, e_2, u_4, e_3, u_3, e_5, u_1, e_1, u_2)$ and $(u_3, e_4, u_2, e_2, u_4, e_3, u_3)$, or
$(u_4, e_3, u_3, e_5, u_1, e_1, u_2, e_2, u_4)$ and $(u_2, e_2, u_4, e_3, u_3, e_4, u_2)$, or
$(u_4, e_3, u_3, e_5, u_1, e_1, u_2, e_2, u_4)$ and $(u_4, e_3, u_3, e_4, u_2, e_2, u_4)$, or
$(u_4, e_3, u_3, e_5, u_1, e_1, u_2, e_2, u_4)$ and $(u_3, e_4, u_2, e_2, u_4, e_3, u_3)$, or
$(u_3, e_5, u_1, e_1, u_2, e_2, u_4, e_3, u_3)$ and $(u_2, e_2, u_4, e_3, u_3, e_4, u_2)$, or
$(u_3, e_5, u_1, e_1, u_2, e_2, u_4, e_3, u_3)$ and $(u_4, e_3, u_3, e_4, u_2, e_2, u_4)$, or
$(u_3, e_5, u_1, e_1, u_2, e_2, u_4, e_3, u_3)$ and $(u_3, e_4, u_2, e_2, u_4, e_3, u_3)$

with cost 11 and 1, respectively.

It is easy to see that each path p' which cannot be further decomposed has less than n^2 vertices, where n is the number of different vertices that occur in p. If we mark in p' one occurrence of each vertex in p, then between two neighboring marked vertices there are at most $n - 1$ unmarked vertices. Otherwise, another simple cycle in p' could be eliminated.

Suppose that a decomposition procedure has exactly removed x_i simple cycles starting at vertex u_i with transit vector t_i and cost value c_i from p. Then, the system

$$(p', (x_1, u_1, t_1, c_1), \ldots, (x_k, u_k, t_k, c_k))$$

is called a *complete path decomposition* of p. Each such path decomposition obeys

$$\text{tran}(p) = \text{tran}(p') + \sum_{i=1}^{k} x_i \cdot t_i$$

and

$$\text{cost}(p) = \text{cost}(p') + \sum_{i=1}^{k} x_i \cdot c_i.$$

Example 4.3. The path p from Example 4.2 has the following twelve complete path decompositions:

$(p', (1, u_1, (-2, 0), 11), (1, u_2, (-5, 3), 1))$, $(p', (1, u_1, (-2, 0), 11), (1, u_4, (-5, 3), 1))$,
$(p', (1, u_1, (-2, 0), 11), (1, u_3, (-5, 3), 1))$, $(p', (1, u_2, (-2, 0), 11), (1, u_2, (-5, 3), 1))$,
$(p', (1, u_2, (-2, 0), 11), (1, u_4, (-5, 3), 1))$, $(p', (1, u_2, (-2, 0), 11), (1, u_3, (-5, 3), 1))$,
$(p', (1, u_4, (-2, 0), 11), (1, u_2, (-5, 3), 1))$, $(p', (1, u_4, (-2, 0), 11), (1, u_4, (-5, 3), 1))$,
$(p', (1, u_4, (-2, 0), 11), (1, u_3, (-5, 3), 1))$, $(p', (1, u_3, (-2, 0), 11), (1, u_2, (-5, 3), 1))$,
$(p', (1, u_3, (-2, 0), 11), (1, u_4, (-5, 3), 1))$, $(p', (1, u_3, (-2, 0), 11), (1, u_3, (-5, 3), 1))$.

LEMMA 4.4. *Let $G = (V, E)$ be a d-dimensional static graph $t_{\max}$ defined as in Fact* 4.1, *$u, v \in V$, and $m \in \mathbb{Z}^d$.*

For each minimum cost m-path from u to v in G, there is a minimum cost m-path from u to v in G that has a complete path decomposition

$$(p', (x_1, u_1, t_1, c_1), \ldots, (x_k, u_k, t_k, c_k))$$

such that

1. $k \leq (2 \cdot t_{\max} \cdot |V| + 1)^d$,
2. *at most d of the x_i's are polynomially bounded in the integers of $t_1, \ldots, t_k$, m, and* $\text{tran}(p')$, *and all remaining x_i's are polynomially bounded only in the integers of $t_1, \ldots, t_k$.*

Proof. Consider any complete path decomposition

$$(p', (x_1, u_1, t_1, c_1), \ldots, (x_k, u_k, t_k, c_k))$$

of a minimum cost m-path p from u to v in G.

1. Without loss of generality we can assume that $t_i = t_j$ implies $c_i = c_j$. Otherwise, p is not of minimal cost. If $t_i = t_j$ and $c_i < c_j$ then we can build a path which passes $x_i + x_j$ cycles with transit t_i and cost c_i at vertex u_i and no cycle with transit t_j and cost c_j at vertex u_j. This new path has $x_j \cdot (c_j - c_i)$ less cost than p. An equivalent argumentation shows that we can assume that all transit vectors t_i are pairwise distinct. By Fact 4.1(2) there are at most $(2 \cdot t_{\max} \cdot |V| + 1)^d$ pairwise different transit vectors for all simple cycles. It follows that there is always a minimum cost m-path that has a complete path decomposition such that

$$k \leq (2 \cdot t_{\max} \cdot |V| + 1)^d.$$

2. To prove the second statement, consider the $d \times k$ matrix A with columns $t_1^T, \ldots, t_k^T$, the column vector $x = (x_1, \ldots, x_k)^T$, and the row vector $c = (c_1, \ldots, c_k)$. By the definition of the path decomposition it follows that

$$\text{tran}(p')^T + A \cdot x = m^T$$

and

$$\text{cost}(p') + c \cdot x = \text{cost}(p).$$

The vector x is an optimal solution of the integer linear program

$$\min \left\{ c \cdot x \;\middle|\; \begin{array}{l} A \cdot x = m^T - \text{tran}(p')^T \\ x \text{ nonnegative integer} \end{array} \right\},$$

because we have decomposed a minimum cost m-path p.

If there is an optimal solution for the program above then there is also one in which all integers are polynomially bounded in the integers of A and the integers of $m^T - \text{tran}(p')^T$ (see [2]). This fact is usually used to show that integer linear programming belongs to NP. Therefore, all integers in x can be assumed to be polynomially bounded in the integers of $t_1, \ldots, t_k, m$, and $\text{tran}(p')$.

The matrix A from above has column rank at most d because the column vectors $t_1^T, \ldots, t_k^T$ have d rows. If A has more than d columns, we can find linear dependences between some of the columns. That is, there exists an integral column vector x' with at most $d + 1$ nonzero entries such that $A \cdot x' = \vec{0}$. Because of [2], the integers in x' can be assumed to be polynomially bounded in the integers of A. If $x + x'$ and $x - x'$ are nonnegative then $c \cdot x' = 0$. This follows from the fact that a minimum cost path is decomposed (see the argumentation in the first part of this proof).

At least $k - d$ of the nonnegative integers in x can be decreased by adding or subtracting certain linear dependences to x. Since all integers in a linear dependence x' are bounded by the integers of A, at least $k - d$ integers in x can be decreased such that they are polynomially bounded in the integers of A. □

Lemma 4.4 also implies that for each fixed dimension d there is a minimum cost m-path in G if and only if there is a minimum cost m-path whose total number of edges is polynomially bounded in the size of G, the integers in the transit vectors, and the integers of m, because the number of four-tuples in a complete path decomposition is exponential only in d. We will not analyze the exact polynomial bound resulting from the path decomposition in Lemma 4.4, because we will show a more general and tighter bound on the number of edges of a minimum cost m-path in Lemma 4.7.

THEOREM 4.5. *Let $G = (V, E)$ be a d-dimensional static graph, $t_{\max}$ be defined as in Fact* 4.1, *$u, v \in V$, and $m \in \mathbb{Z}^d$ for some constant d.*

If $\text{min\_cost}(G, u, v, m) \neq -\infty$, *then* $\text{min\_cost}(G, u, v, m)$ *is computable in polynomial time with respect to* $\text{size}(G)$, $t_{\max}$, *and* $\text{size}(m)$.

Proof. Assume there exists a minimum cost m-path from u to v in G. Then by Lemma 4.4, we know that there is a minimum cost m-path p from u to v that has a complete path decomposition

$$(p', (x_1, u_1, t_1, c_1), \ldots, (x_k, u_k, t_k, c_k))$$

such that $k \leq (2 \cdot t_{\max} \cdot |V| + 1)^d$. Additionally, we know that at most d of the x_i's are polynomially bounded in $\text{size}(G)$, $t_{\max}$, and $\|m\|$, and all other x_i's are polynomially bounded

in size(G) and t_{max}. This follows from Fact 4.1(1) and because p' has at most $|V|^2$ edges and each cycle removed during a decomposition is simple. If such a decomposition does not exist, there is no m-path from u to v in G. Note also that p' and all removed simple cycles are of minimal cost, because each subpath of a minimum cost path is a minimum cost path. We call the systems (x_i, u_i, t_i, c_i), where the x_i's are polynomially bounded only in size(G) and t_{max}, the *small* systems.

Each minimum cost m-path p can be decomposed into

1. a minimum cost path p'' with a number of edges that is polynomially bounded in size(G) and t_{max}, and
2. at most $l \leq d$ simple cycles with transit vectors $t_1, \ldots, t_l$ such that there are some positive integers $y_1, \ldots, y_l$ with

$$\sum_{i=1}^{l} y_i \cdot t_i = m - \text{tran}(p'')$$

and

$$\sum_{i=1}^{l} y_1 \cdot c_i = \text{cost}(p) - \text{cost}(p'').$$

The path p'' is the path p' together with all the simple cycles represented by the small systems.

Now determine

1. all sets $U \subseteq V$ of $l \leq d$ vertices,
2. all choices $t_1, c_1, \ldots, t_l, c_l$ of l transit vectors and cost values of simple minimum cost cycles starting at vertices from U, and
3. all transit vectors t'' and cost values c'' of minimum cost paths p'' from u to v with at most $\hat{n}$ edges containing all vertices from U, where $\hat{n} = |V|^2$ (the maximal number of edges in p') $+\hat{x} \cdot |V|$ (the maximal number of edges of all cycles of the small systems), where $\hat{x}$ is a polynomial upper bound for the x_i's in the small systems,

such that there are nonnegative integers $y_1, \ldots, y_l$ such that

$$\sum_{i=1}^{l} y_i \cdot t_i = m - \text{tran}(p'').$$

The number of all these choices is polynomially bounded in size(G) and t_{max}, because for each fixed d

1. the number of subsets U is polynomially bounded in $|V|$ because l is fixed,
2. the number of different transit vectors and cost values for simple minimum cost cycles is polynomially bounded in size(G) and t_{max} (see Fact 4.1(2)),
3. the number of different transit vectors and cost values for all minimum cost paths p'' with at most $\hat{n}$ edges that pass all vertices from U is polynomially bounded in size(G) and t_{max} (see Fact 4.1(2)).

It is also possible to generate all these sets U, transit vectors $t_1, \ldots, t_l$ and cost values $c_1, \ldots, c_l$ for the cycles, and transit vectors t'' and cost values c'' for the paths p'' in polynomial time with respect to size(G) and t_{max} (see Fact 4.1(3) and 4.1(4)). The nonnegative integers $y_1, \ldots, y_l$ that yield a minimal cost value

$$\sum_{i=1}^{l} y_i \cdot c_i$$

are computable in polynomial time with respect to size(G) and size(m), because integer linear programs with a fixed number of variables are solvable in polynomial time [11]. The cost of a minimum cost m-path is the minimal cost value

$$c'' + \sum_{i=1}^{l} y_i \cdot c_i$$

such that

$$t'' + \sum_{i=1}^{l} y_i \cdot t_i = m$$

for all choices considered above. □

The proof of Theorem 4.5 also shows that we can find an m-path in polynomial time with respect to size(G), $t_{\max}$, and size(m), if one exists. The same idea can be used to prove the following corollary.

COROLLARY 4.6. *Let $G = (V, E)$ be a d-dimensional static graph, $t_{\max}$ be defined as in Fact* 4.1, *$u, v \in V$, and $m \in \mathbb{Z}^d$ for some constant d.*

For each set $W \subseteq V$ with a constant number of vertices, it is decidable in polynomial time with respect to size(G), $t_{\max}$, *and* size(m) *whether there is an m-path from u to v in G that passes all vertices in W.*

Proof. The proof proceeds analogously to the argumentation in the proof of Theorem 4.5 with the extension that p'' additionally passes all vertices from W (see Fact 4.1(4)). □

We continue with the problem of deciding whether there are infinitely many m-paths with decreasing costs.

LEMMA 4.7. *Let $G = (V, E)$ be a d-dimensional static graph, $t_{\max}$ be defined as in Fact* 4.1, *$u, v \in V$, and $m \in \mathbb{Z}^d$.*

Then, min_cost(G, u, v, m) *is $-\infty$ if and only if there is an m-path p from u to v that has a complete path decomposition*

$$(p', (x_1, u_1, t_1, c_1), \ldots, (x_k, u_k, t_k, c_k))$$

such that there are nonnegative integers $y_1, \ldots, y_k$ such that

$$\sum_{i=1}^{k} y_i \cdot t_i = 0$$

and

$$\sum_{i=1}^{k} y_i \cdot c_i < 0,$$

and at most $l \leq d + 1$ of the y_i's are nonzero.

Proof. $\Leftarrow$ The cost of the m-path p can be decreased below any bound by extending p at vertex u_i with y_i simple cycles with transit vector t_i and cost c_i for $i = 1, \ldots, k$.

$\Rightarrow$ If min_cost(G, u, v, m) $= -\infty$, there must be infinitely many m-paths from u to v in G with different negative costs. But for each graph G the number of paths p', vertices u_i, transit vectors t_i, and cost values c_i in all possible complete path decompositions

$$(p', (x_1, u_1, t_1, c_1), \ldots, (x_k, u_k, t_k, c_k))$$

of m-paths in G is finite. If we were to construct for each such m-path a complete path decomposition, then there would be an infinite number of path decompositions which differ only in the x_i's. Since all x_i's are positive, a simple counting argument reveals that there must exist two such complete path decompositions

$$(p', (x_1, u_1, t_1, c_1), \ldots, (x_k, u_k, t_k, c_k))$$

and

$$(p', (x_1', u_1, t_1, c_1), \ldots, (x_k', u_k, t_k, c_k))$$

such that $x_i \leq x_i'$ for $1 \leq i \leq k$ and

$$\sum_{i=1}^{k} x_i \cdot c_i > \sum_{i=1}^{k} x_i' \cdot c_i.$$

Since all paths are m-paths, there must exist some nonnegative integers $y_1 (= x_1' - x_1), \ldots, y_k (= x_k' - x_k)$ such that

$$\sum_{i=1}^{k} y_i \cdot t_i = \vec{0}$$

and

$$\sum_{i=1}^{k} y_i \cdot c_i < 0.$$

As in the proof of Lemma 4.4, consider the $d \times k$ matrix A with columns $t_1^T, \ldots, t_k^T$, the column vector $y = (y_1, \ldots, y_k)^T$, and the row vector $c = (c_1, \ldots, c_k)$. We know that $A \cdot y = \vec{0}$ and $c \cdot y < 0$. Obviously, the linear system of equations $A \cdot y = \vec{0}$ and inequalities $c \cdot y < 0$ has a nonnegative *rational* solution such that at most $d + 1$ entries of y are nonzero. The extension of y to a nonnegative integral solution of $A \cdot y = \vec{0}$ preserves $c \cdot y < 0$. □

Lemma 4.7 implies the polynomial time decidability of the existence of an unbounded solution, as the next theorem shows.

THEOREM 4.8. *Let $G = (V, E)$ be a d-dimensional static graph, $t_{\max}$ be defined as in Fact* 4.1, *$u, v \in V$, and $m \in \mathbb{Z}^d$ for some constant d.*

Then, the question whether min_cost(G, u, v, m) *is $-\infty$ is decidable in polynomial time with respect to* size(G), $t_{\max}$, *and* size(m).

Proof. Corollary 4.6 implies that for each fixed dimension d and each vertex set W of at most $d + 1$ vertices, it is possible to test in polynomial time with respect to size(G), $t_{\max}$, and size(m) whether there is an m-path from u to v in G passing all vertices in W. That is, we can consider

1. all sets $W \subseteq V$ of $l \leq d + 1$ vertices and
2. all choices $t_1, c_1, \ldots, t_l, c_l$ of l transit vectors and cost values of simple minimum cost cycles starting at vertices from W

such that there exists an m-path from u to v passing all vertices in W, and there are some nonnegative integers $y_1, \ldots, y_l$ with

$$\sum_{i=1}^{l} y_i \cdot t_i = 0$$

and

$$\sum_{i=1}^{l} y_i \cdot c_i < 0.$$

The number of all these choices is again polynomially bounded in size(G) and $t_{\max}$. The negative cost condition can even be verified in polynomial time with respect to size(G), because linear programming belongs to P. □

The main result of this section follows from Theorems 4.5 and 4.8.

COROLLARY 4.9. *Let $G = (V, E)$ be a d-dimensional static graph, $t_{\max}$ be defined as in Fact* 4.1, *$u, v \in V$, and $m \in \mathbb{Z}^d$ for some constant d.*

Then, min_cost(G, u, v, m) *is computable in polynomial time with respect to size(G), $t_{\max}$, and size(m).*

Proof. First decide whether min_cost(G, u, v, m) is $-\infty$ as in the proof of Theorem 4.5. If this does not hold, compute min_cost(G, u, v, m) as in the proof of Theorem 4.8. □

We have shown that the MINIMUM COST m-PATH problem for fixed dimensions can be solved in polynomial time with respect to the integers in the transit vectors. The solution method introduced in the theorems above does not always yield an interesting algorithm from a practical point of view. This is because the number of (small) integer linear programs that we have to solve is exponential in d. However, it asymptotically improves the straightforward solution of solving $2^{|E|}$ different integer linear programs as considered in Theorem 3.2 for all choices of subsets of edges.

5. Bounds on the path-length. Now we prove an upper bound on the total number of edges in a minimum cost m-path. This bound can, for example, be used to control algorithms for finding schedule functions. The proof is based on the following theorem of Grinberg and Sevastýanov [5, Thm. 1].

THEOREM 5.1 [5]. *Let $x_1, \ldots, x_N, x \in \mathbb{R}^d$ be any collection of d-dimensional vectors such that $\sum_{i=1}^{N} x_i = x$ and $\|x_i\| \leq 1$ for $i = 1, \ldots, N$. Then there is a permutation π of the integers $\{1, \ldots, N\}$ such that*

$$\left\| \sum_{i=1}^{j} x_{\pi(i)} - \frac{j-d}{N} \cdot x \right\| \leq d$$

for $j = 1, \ldots, N$.

This theorem can be used to show the following lemma.

LEMMA 5.2. *Let $t_1, \ldots, t_N, t \in \mathbb{Z}^d$ be a collection of d-dimensional integral vectors such that $\sum_{i=1}^{N} t_i = t$ and $\|t_i\| \leq r$ for some integer r and $i = 1, \ldots, N$. If, for each nonempty subset $A \subseteq \{t_1, \ldots, t_N\}$, it holds that $\sum_{t_k \in A} t_k \neq \vec{0}$, then*

$$N \leq (2 \cdot r \cdot d + 1)^d \cdot \|t\|_2,$$

where $\|t\|_2$ is the Euclidean norm.[3]

Proof. By Theorem 5.1 it follows that there is a permutation π of the integers $\{1, \ldots, N\}$ such that

$$\left\| \sum_{i=1}^{j} t_{\pi(i)} - \frac{j-d}{N} \cdot t \right\| \leq r \cdot d$$

[3] $\|y\|_2 = (\sum_{i=1}^{d} y_i^2)^{(1/2)}$ for $y = (y_1, \ldots, y_d) \in \mathbb{R}^d$.

for $j = 1, \ldots, N$. This inequality says that the sum of the first j vectors $\sum_{i=1}^{j} t_{\pi(i)}$ differs from vector $\frac{j-d}{N} \cdot t$ by at most $r \cdot d$ in each position. There are at most $(2 \cdot r \cdot d + 1)^d$ integral points which differ from some point $\frac{j-d}{N} \cdot t$ by at most $r \cdot d$ in each position. Now we estimate the number of integral points which differ from some point of the set $\{\frac{j-d}{N} \cdot t \mid j := 1, \ldots, N\}$ by at most $r \cdot d$ in each position. This number of integral points is roughly estimated by

$$(2 \cdot r \cdot d + 1)^d \cdot \|t\|_2.$$

Since for each nonempty subset $A \subseteq \{t_1, \ldots, t_N\}$ all sums $\sum_{t_k \in A} t_k$ are nonzero, the number of integral points estimated above is an upper bound on the number of integral vectors which are added in the complete sum. □

Now we give an upper bound on the total number of edges in a minimum cost m-path.

LEMMA 5.3. *Let $G = (V, E)$ be a d-dimensional static graph, $t_{\max}$ be defined as in Fact 4.1, $u, v \in V$, and $m \in \mathbb{Z}^d$.*

If there exists a minimum cost m-path in G from u to v, then there also exists a minimum cost m-path in G from u to v with at most

$$|V|^2 + |V| \cdot (2 \cdot |V| \cdot t_{\max} \cdot d + 1)^d \cdot \|m - tran(p')\|_2$$

edges.

Proof. Let p be a minimum cost m-path in G from u to v with a minimal number of edges. Let

$$(p', (x_1, u_1, t_1, c_1), \ldots, (x_k, u_k, t_k, c_k))$$

be a complete path decomposition for p and let $N = \sum_{i=1}^{k} x_i$. By the definition of a path decomposition and by Fact 4.1, we know that

- p' has at most $|V|^2$ edges,
- $\|\text{tran}(p')\| \leq |V|^2 \cdot t_{\max}$, and
- $\|t_i\| \leq |V| \cdot t_{\max}$ for all t_i's.

Without loss of generality we can assume that all vector subsums are different, because p has minimal cost and a minimal number of edges. For $r = |V| \cdot t_{\max}$ and $t = m - \text{tran}(p')$, by Lemma 5.2 we obtain the inequality

$$N \leq (2 \cdot |V| \cdot t_{\max} \cdot d + 1)^d \cdot \|m - \text{tran}(p')\|_2.$$

Since each t_i originates from a cycle with at most $|V|$ edges and the path p' has at most $|V|^2$ edges, the path p has at most

$$|V|^2 + |V| \cdot (2 \cdot |V| \cdot t_{\max} \cdot d + 1)^d \cdot \|m - \text{tran}(p')\|_2$$

edges. □

Lemma 5.3 also generalizes and improves a result given by Orlin in [13, Lem. 6] for one-dimensional reachability problems.

Note that the bound of Lemma 5.3 cannot be used to solve the unboundedness problem of minimum cost m-paths, i.e., it cannot be used to solve the problem of whether for each rational b there is an m-path with cost less than b. However, a necessary condition for the unboundedness of minimum cost m-paths is the existence of a nonnegative rational solution of the system $A \cdot x = \vec{0}$ with $c \cdot x < 0$, where A and c are the vertex-edge adjacency matrix and cost vector as defined for the general solution in §3. Note that this condition is not sufficient, because we do not know whether there is any m-path containing all cycles represented by such

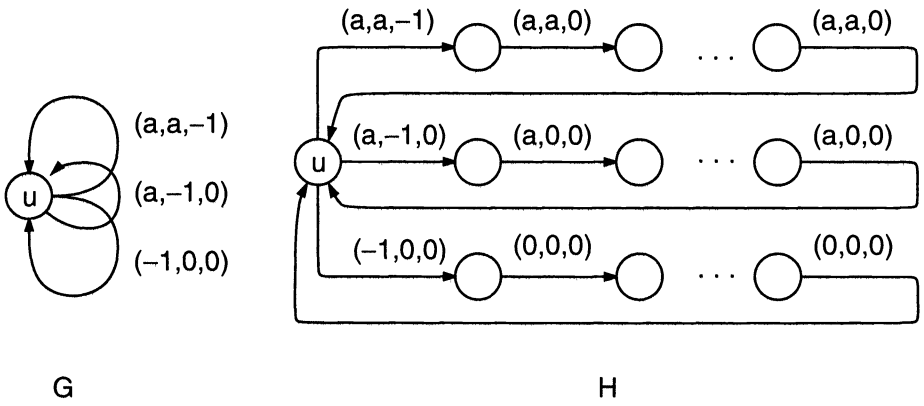


FIG. 5.1. *Two three-dimensional reduced dependence graphs.*

a solution. For many interesting situations, however, the absence of m-paths with decreasing costs can be presupposed as, for example, in static graphs without negative costs or without $\vec{0}$-cycles.

The next lemma proves the existence of minimum cost m-paths with an exponential number of edges.

LEMMA 5.4. *There are d-dimensional static graphs $G = (V, E)$ that have minimum cost m-paths with*

$$\Omega\left(\left(t_{\max} \cdot \frac{|V|}{d}\right)^{d-1} \cdot \|m\|_2\right)$$

edges, where $t_{\max}$ is defined as in Fact 4.1.

Proof. Let a be some integer and G be a d-dimensional static graph with one vertex u and d edges from u to u with transit vectors

$$\begin{aligned}
&(a, \ldots, a, a, a, -1),\\
&(a, \ldots, a, a, -1, 0),\\
&(a, \ldots, a, -1, 0, 0),\\
&\qquad\vdots\\
&(a, a, -1, 0, 0, \ldots, 0),\\
&(a, -1, 0, 0, 0, \ldots, 0),\\
&(-1, 0, 0, 0, 0 \ldots, 0).
\end{aligned}$$

Any $(0, \ldots, 0, -b \cdot a)$-path from u to u for any b has

$$\Omega(a^d \cdot b)$$

edges.

Let H be the graph consisting of d cycles with one common vertex u such that each cycle has the same number of edges labeled as in the example of Fig. 5.1 for $d = 3$. Then, any $m = (0, \ldots, 0, -b \cdot a \cdot (\frac{|V|-1}{d} + 1))$-path from u to u in H has $\Omega((a \cdot \frac{|V|}{d})^d \cdot b)$ edges or, equivalently,

$$\Omega\left(\left(t_{\max} \cdot \frac{|V|}{d}\right)^{d-1} \cdot \|m\|_2\right)$$

edges. □

Acknowledgments. We thank Thomas Lengauer, who has initiated, motivated, and assisted our research concerning algorithms for graph problems on periodic graphs.

REFERENCES

[1] W. BACKES, U. SCHWIEGELSHOHN, AND L. THIELE, *Analysis of free schedule in periodic graphs*, in Proc. Annual ACM Symposium on Parallel Algorithms and Architectures, Association for Computing Machinery, 1992, pp. 333–343.

[2] I. BOROSH AND L. TREYBIG, *Bounds on positive integral solutions of linear diophantine equations*, Proc. Amer. Math. Soc., 55 (1976), pp. 299–304.

[3] E. COHEN AND N. MEGIDDO, *Strongly polynomial-time and NC algorithms for detecting cycles in dynamic graphs*, J. Assoc. Comput. Mach., 40 (1993), pp. 791–830.

[4] ———, *Recognizing properties of periodic graphs*, in Applied Geometry and Discrete Mathematics. The Victor Klee Festschrift, Vol. 4, P. Gritzmann and B. Sturmfels, eds., Association for Computing Machinery, 1991, pp. 135–146.

[5] V. GRINBERG AND S. SEVASTÝANOV, *Value of the Steinitz constant*, Funct. Anal. Appl., 14 (1980), pp. 125–126.

[6] K. IWANO AND K. STEIGLITZ, *Testing for cycles in infinite graphs with periodic structure*, in Proc. Annual ACM Symposium on Theory of Computing, New York, NY, 1987, pp. 46–55.

[7] R. KARP, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, R. Miller and J. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–103.

[8] R. KARP, R. MILLER, AND A. WINOGRAD, *The organization of computations for uniform recurrence equations*, J. Assoc. Comput. Mach., 14 (1967), pp. 563–590.

[9] M. KODIALAM AND J. ORLIN, *Recognizing strong connectivity in (dynamic) periodic graphs and its relation to integer programming*, in Proc. ACM–SIAM Symposium on Discrete Algorithms, 1991, pp. 131–135.

[10] S. KOSARAJU AND G. SULLIVAN, *Detecting cycles in dynamic graphs in polynomial time*, in Proc. Annual ACM Symposium on Theory of Computing, Chicago, IL, 1988, pp. 398–406.

[11] H. LENSTRA, *Integer programming with a fixed number of variables*, Math. Oper. Res., 8 (1983), pp. 538–548.

[12] G. NEMHAUSER AND L. WOLSEY, *Integer and Combinatorial Optimization*, in Wiley-Interscience Series in Discrete Mathematics and Optimization, John Wiley, New York, 1989.

[13] J. ORLIN, *Some problems on dynamic/periodic graphs*, in Progress in Combinatorial Optimization, W. Pulleyblank, ed., Academic Press, Orlando, FL, 1984, pp. 273–293.

[14] C. PAPADIMITRIOU, *On the complexity of integer programming*, J. Assoc. Comput. Mach., 28 (1981), pp. 765–768.

[15] S. RAO, *Regular iterative algorithms and their implementations on processor arrays*, Ph.D. thesis, Department of Electrical Engineering, Stanford University, 1985.

[16] A. SCHRIJVER, *Theory of linear and integer programming*, in Wiley-Interscience Series in Discrete Mathematics and Optimization, John Wiley, New York, 1986.

SIAM J. COMPUT.
Vol. 24, No. 5, pp. 1068-1081, October 1995

008

POLYNOMIAL-TIME MEMBERSHIP COMPARABLE SETS*

MITSUNORI OGIHARA†

Abstract. This paper studies a notion called polynomial-time membership comparable sets. For a function g, a set A is polynomial-time g-membership comparable if there is a polynomial-time computable function f such that for any $x_1, \ldots, x_m$ with $m \geq g(\max\{|x_1|, \ldots, |x_m|\})$, f outputs $b \in \{0, 1\}^m$ such that $(A(x_1), \ldots, A(x_m)) \neq b$. The following is a list of major results proven in the paper:

1. Polynomial-time membership comparable sets construct a proper hierarchy according to the bound on the number of arguments.
2. Polynomial-time membership comparable sets have polynomial-size circuits.
3. For any function f and any constant $c > 0$, if a set is $\leq^p_{f(n)\text{-}tt}$-reducible to a P-selective set, then the set is polynomial-time $(1 + c) \log f(n)$-membership comparable.
4. For any $\mathcal{C}$ chosen from {PSPACE, UP, FewP, NP, $C_=P$, PP, MOD_2P, MOD_3P, ...}, if $\mathcal{C} \subseteq$ P-mc($c \log n$) for some $c < 1$, then $\mathcal{C} =$ P.

As a corollary of the last two results, it is shown that if there is some constant $c < 1$ such that all $\mathcal{C}$ are polynomial-time n^c-truth-table reducible to some P-selective sets, then $\mathcal{C} =$ P, which resolves a question that has been left open for a long time.

Key words. P-selective sets, polynomial-time reducibilities, polynomial-size circuits

AMS subject classifications. 68Q15, 68Q05

1. Introduction.

Given two strings x and y, can we tell which is more likely to be in a set A? Jockusch [20] defined a set A as *semirecursive* if there is a recursive function f such that for all x and y, (i) $f(x, y) \in \{x, y\}$ and (ii) if $\{x, y\} \cap A \neq \emptyset$, then $f(x, y) \in A$. We call the function f a *selector* for A. Selman [30] considered a polynomial-time analogue of semirecursive sets and defined a set A as *P-selective* if A has a polynomial-time computable selector. P-selective sets have been widely studied [3], [11], [16]–[18], [22], [25], [27], [30]–[32], [36]. Recently, there have been some remarkable results about P-selective sets. Buhrman, van Helden, and Torenvliet [11] have shown that a set is in P if and only if it is $\leq^p_T$-self-reducible and P-selective, while previously known characterization is $A \in$ P if and only if A is $\leq^p_{ptt}$-self-reducible and P-selective [32]. By constructing P-selective sets with certain properties, Hemaspaandra et al. [17] and Naik, Ogiwara, and Selman [27] have proven that NP search problems are not reducible to corresponding decision problems unless some implausible collapses of exponential-time complexity classes occur. Hemachandra et al. [16] studied internal structure of the class of sets $\leq^p_T$-reducible to P-selective sets and introduced the notion of $\mathcal{FC}$-selectivity for various function classes $\mathcal{FC}$. Hemaspaandra et al. [18] have studied sets with nondeterministically polynomial-time computable selectors, and have proven that if there is an NP-function that computes satisfying assignments uniquely, then the polynomial-time hierarchy [26], [34], collapses to its second level Σ^p_2.

For a set A, let us identify A and its characteristic function. For any x and y, there are four possible values of $(A(x), A(y))$. By mapping a pair (x, y) to y, a selector for A declares that $x \in A \rightarrow y \in A$ or, equivalently, $(A(x), A(y)) \neq (1, 0)$. Thus one can view a selector for A as a function f that maps (x, y) to $b \in \{0, 1\}^2$ such that $(A(x), A(y)) \neq b$, where b is always either 01 or 10. An interesting and fundamental question arising from this observation is "how strong is the restriction $b \in \{01, 10\}$?" That is, if we allow f to map to 00 or 11, then how

*Received by the editors November 5, 1993; accepted for publication (in revised form) May 9, 1994. This work was supported in part by National Science Foundation and Japan Society for the Promotion of Science grant NSF-INT-9116781/JSPS-ENG-207. This work was done in part while the author was at the University of Electro-Communications, Chofu-shi, Tokyo, Japan.

†Department of Computer Science, University of Rochester, Rochester, NY 14627 (ogihara@cs.rochester.edu).

much does the complexity of A increase? Let us consider this in a more generalized setting and define the notion of polynomial-time membership comparable sets. Let $\mathbf{N}$ ($\mathbf{N}^+$) denote the set of all nonnegative (positive) integers. Call a function $g : \mathbf{N} \to \mathbf{N}^+$ *polynomially bounded* if there is a polynomial p such that for every n, $g(n) \leq p(n)$, and call it *polynomial-time computable* if there is a polynomial-time bounded machine that, on input x, outputs $1^{g(|x|)}$.

DEFINITION 1.1. *Let $g : \mathbf{N} \to \mathbf{N}^+$ be monotone nondecreasing, polynomial-time computable, and polynomially bounded.*

1. *A function f is called a g-membership comparing function (a g-mc-function) for A if for every $x_1, \ldots, x_m$ with $m \geq g(\max\{|x_1|, \ldots, |x_m|\})$,*

$$f(x_1, \ldots, x_m) \in \{0, 1\}^m \text{ and } (A(x_1), \ldots, A(x_m)) \neq f(x_1, \ldots, x_m).$$

2. *A set A is said to be polynomial-time g-membership comparable if there exists a polynomial-time computable g-mc-function for A.*

3. P-mc(g) *denotes the class of all polynomial-time g-membership comparable sets.*

A crucial property of mc-functions is that they exclude one value out of 2^m possible values. The notion—excluding possible values of $(A(x_1), \ldots, A(x_m))$—has already appeared in the literature. For a fixed $k \geq 1$, the function that on input $x_1, \ldots, x_k$, outputs $A(x_1) \ldots A(x_k)$ has been called F_k^A; some notions related to F_k^A have been introduced and studied (see [5], [4], [7], [8]). If one can always reduce 2^k possible values of F_k^A to $m < 2^k$, then F_k^A is said to be computable by a set of m polynomial-time functions [4] and m-enumerable [12]. A set A is non-p-superterse [7] if for some $k \geq 1$, there is a polynomial-time algorithm that computes F_k^A using $k - 1$ adaptive queries to some set X. So, for a non-p-superterse set A, F_k^A is 2^{k-1}-enumerable for some $k \geq 1$. Polynomial-time membership comparable sets are more general than these notions in the sense that (i) k can be increased according to the length of the input and (ii) only one value is required to be excluded.

When a function g with the set of real numbers as its range is used, we will be identifying g and $\lambda n.[\max\{1, \lfloor g(n) \rfloor\}]$ purely as a convention. We use P-mc(const), P-mc(log), and P-mc(poly), respectively, to denote $\cup\{\text{P-mc}(k) : k \geq 1\}$, $\cup\{\text{P-mc}(f) : f = \mathcal{O}(\log n)\}$, and $\cup\{\text{P-mc}(p) : p \text{ is a polynomial}\}$, where log is base 2. It is possible for many different (indeed, even an uncountable number of) sets to be in P-mc(poly) via the same function g. See Proposition 4.1 for an example of when this actually happens.

In §2 we study basic properties of polynomial-time membership comparable sets. It is easily observed that the smallest P-mc class, namely P-mc(1), is equal to P. Noting that P-Sel, the class of P-selective sets, is a subclass of P-mc(2) and that P $\subset$ P-Sel [30], we have P-mc(1) $\subset$ P-mc(2). We show that the inequality holds for arbitrary k; that is, P-mc(k) $\subset$ P-mc($k + 1$) for any $k \geq 1$. More generally, we prove that for any f and g such that $f(n) < g(n)$ for finitely many n, P-mc(g) contains a set not in P-mc(f). Therefore, P-mc classes construct a proper hierarchy according to the bound on the number of arguments.

Because P-mc(2) sets can be viewed as less restrictive P-selective sets, one might expect that P-mc sets do not go far beyond P-Sel. In §3 we consider the question of how they are related to each other. We seek to prove inclusions between reducibility classes of P-Sel and P-mc classes. For a reducibility $\leq_r^p$ and a class $\mathcal{C}$, let $R_r(\mathcal{C})$ denote the class of all sets that are $\leq_r^p$-reducible to some set in $\mathcal{C}$. Basically, P-mc(2) properly includes P-Sel: there is a tally set in P-mc(2) $-$ P-Sel. Furthermore, we prove for any function f that $R_{f(n)\text{-}tt}(\text{P-Sel}) \subseteq$ P-mc($(1 + c) \log f(n)$) for any constant $c > 0$, which yields $R_{btt}(\text{P-Sel}) \subseteq$ P-mc(const) and $R_{tt}(\text{P-Sel}) \subseteq$ P-mc(log).

We also study the question of the other direction, namely, whether P-mc sets are polynomial-time reducible to P-selective sets. We show that P-mc(poly) $\subseteq$ P/poly, where P/poly is the class of all sets having polynomial-size circuits. Then, since $R_T(\text{P-Sel}) =$

P/poly [22], we have P-mc(poly) $\subseteq R_T$(P-Sel). The $\leq^p_T$-reducibility in this inclusion is optimal because we show R_{tt}(P-Sel) $\subset$ P-mc(poly). On the other hand, the converse statement P/poly $\subseteq$ P-mc(poly) does not appear to hold. But we can show at least that the question is very subtle because P/poly $\not\subseteq$ P-mc(poly) implies P $\neq$ NP. Thus, proving that P-mc(poly) $\subset$ P/poly is at least as hard as proving P $\neq$ NP. We conjecture that the converse inclusion does not hold.

It is well known that P/poly $= R_{tt}$(SPARSE) $= R_{tt}$(TALLY), where SPARSE (TALLY) denotes the class of all sparse (tally) sets. Because it holds that P-mc(poly) $\subseteq$ P/poly, every P-mc set is $\leq^p_{tt}$-reducible to some tally set. In §4 we attempt to find close relationships between P-mc sets and sparse sets as well as tally sets. We prove that $R_{1\text{-}tt}$(SPARSE) $\subseteq$ P-mc(poly), and thus R_{tt}(P-mc(poly)) $=$ P/poly. Interestingly, this contrasts with R_{tt}(P-Sel) $\subset$ P/poly [16]. In order to study relationships between P-mc sets and tally sets, binary real numbers will be useful, since the characteristic sequence of a tally set can be viewed as a binary real number in [0, 1) (see [30]). For any binary real number $r \in [0, 1)$, we show that Prefix$[r]$, the set of all prefixes of r, is in P-mc(2), while there is a binary real number r such that Prefix$[r] \notin$ P-Sel.

We add a few words about relationships between P-mc(poly) and TALLY. Noting that P-mc(poly) $\subseteq$ P/poly and TALLY $\subseteq$ P-mc(poly), for every $A \in$ P-mc(poly) we show that there is a tally set T in P-mc(poly) $\cap\, \Delta^p_3(A)$ such that $A \leq^p_{tt} T$. On the other hand, any tally set is $\leq^p_T$-equivalent to some P-mc(2) set. Thus, every set A in P-mc(poly) is $\leq^p_T$-reducible to some set in P-mc(2)$\cap\, \Delta^p_3(A)$. As a consequence, we have P/poly $= R_T$(SPARSE$\cap$P-mc(2)); that is, $\leq^p_T$-reducibility to sparse sets in P-mc(2) completely characterizes P/poly.

Selman [30] showed that SAT is P-selective if and only if SAT is in P. Can we prove a similar result for P-mc sets? We consider this question in §5. Noting that P-mc(poly) $\subseteq$ P/poly and that NP $\subseteq$ P/poly implies PH $= \Sigma^p_2$ [21], one can easily observe that SAT $\in$ P-mc(poly) only if PH $= \Sigma^p_2$. But we have a stronger collapse for some P-mc class. We prove NP $\subseteq$ P-mc($c \log n$) for some $c < 1$ if and only if P $=$ NP.

The proof technique we develop enables us to resolve some open questions. The first bonus we get is the following result: If NP $\subseteq R_{n^c\text{-}tt}$(P-Sel) for some constant $c < 1$, then P $=$ NP. One of the most important questions has been whether NP $\subseteq R_r$(P-Sel) $\Longrightarrow$ P $=$ NP holds for a reducibility $\leq^p_r$ [32], [36], [16], [35]. Selman extended his first result to $\leq^p_{ptt}$-reducibility by observing that R_{ptt}(P-Sel) $= R_m$(P-Sel) [32]. As for "nonpositive" truth-table reducibilities, there have appeared some observations. Toda [36] proved that NP $\subseteq R_{tt}$(P-Sel) implies P $=$ FewP and RP $=$ NP. Hemachandra et al. [16] noticed that NP $\subseteq R_{1\text{-}tt}$(P-Sel) implies P $=$ NP. Thierauf, Toda, and Watanabe [35] proved that if NP $\subseteq R_{btt}$(P-Sel), then NP is in deterministic subexponential time. Nonetheless, the question of whether NP $\subseteq R_{tt}$(P-Sel) implies P $=$ NP, and even whether NP $\subseteq R_{btt}$(P-Sel) implies P $=$ NP, has been open for a long time. We not only give an affirmative answer to the latter, but improve the upper bound on the number of queries to n^c for any constant $c < 1$. We note here that the same result has been independently proven by Beigel, Kummer, and Stephan [9] and Agrawal and Arvind [1].

Another bonus we get concerns the complexity of functions that are polynomial-time computable with access to sets in NP. Krentel [23] showed that for any function $f(n) \leq \frac{1}{2} \log n$, if $\mathrm{FP}^{\mathrm{SAT}}_{f(n)\text{-}T} = \mathrm{FP}^{\mathrm{SAT}}_{(f(n)-1)\text{-}T}$, then P $=$ NP. He asked whether the same statement holds for a larger function f. Krentel's proof directly applies to the case $f(n) \leq c \log n$ for some constant $c < 1$. Beigel [6] asked the related question of whether $\mathrm{FP}^{\mathrm{SAT}}_{f(n)\text{-}tt} \subseteq \mathrm{FP}^{\mathrm{SAT}}_{(f(n)-1)\text{-}T}$. For the case $f(n) \leq c \log n$ with $c < 1$, Beigel [6] showed that for any $\leq^p_{1\text{-}tt}$-hard set A for NP, if $\mathrm{FP}^{A}_{f(n)\text{-}tt} \subseteq \mathrm{FP}^{X}_{(f(n)-1)\text{-}T}$ for some X, then RP $=$ NP and P $=$ UP. Regarding the general $\mathcal{O}(\log n)$ case, Amir, Beigel, and Gasarch [4] showed that for any function $f(n) = \mathcal{O}(\log n)$, if $\mathrm{FP}^{\mathrm{SAT}}_{f(n)\text{-}tt} \subseteq \mathrm{FP}^{X}_{(f(n)-1)\text{-}T}$ for some X, then $\Sigma^p_3 = \Pi^p_3$. In this paper, we prove that the conclusion of Beigel's result can be strengthened to P $=$ NP.

The proof techniques we develop can be applied to complexity classes other than NP. For any class $\mathcal{C}$ chosen from {UP, FewP, $C_=P$, PP, PSPACE, MOD_2P, MOD_3P, ...}, we prove that if $\mathcal{C} \subseteq$ P-mc($c \log n$) for some $c < 1$, then $\mathcal{C} =$ P; and thus, if $\mathcal{C} \subseteq R_{n^c\text{-}tt}$(P-Sel) for some $c < 1$, then $\mathcal{C} =$ P.

2. Basic properties of polynomial-time membership comparable sets. In this section, we prove some fundamental properties of polynomial-time membership comparable sets. The reducibility notions we will use are from [24]. First we state some rather trivial properties.

PROPOSITION 2.1. 1. P = P-mc(1).

2. *For any f and g such that $f(n) \leq g(n)$ for all but finitely many n,* P-mc(f) $\subseteq$ P-mc(g).

3. *If $A \leq^p_{1\text{-}tt} B$ and B is in* P-mc($f(n)$), *then $A \in$* P-mc($f(p(n))$) *for some polynomial p. Especially, for any $k \geq 1$,* P-mc(k) *is closed under $\leq^p_{1\text{-}tt}$-reductions.*

Proof. 1. The statement holds because for any set A, A is in P if and only if A^c is in P if and only if there is a polynomial-time computable function f such that for all x, $f(x) = A^c(x)$.

2. Let f and g be as in the hypothesis and let n_0 be such that for all $n > n_0$, $f(n) \leq g(n)$. Let $A \in$ P-mc(f) and let T be the set of all strings in A of length at most n_0. Let $x_1, \ldots, x_m$ and n be such that $\max\{|x_1|, \ldots, |x_m|\} = n$ and $m \geq g(n)$. If there is some i such that $|x_i| \leq n_0$, then $(A(x_1), \ldots, A(x_m)) \neq 0^{i-1}b0^{m-i-1}$, where $b = 0$ if $x_i \in T$ and 1 otherwise. If, for every i, $|x_i| > n_0$ then $(A(x_1), \ldots, A(x_m)) \neq f(x_1, \ldots, x_{f(n)})0^{m-f(n)}$. Therefore, A is in P-mc(g).

3. Let $A \leq^p_{1\text{-}tt} B$ via a machine M and $B \in$ P-mc(f) via h. Let p be a polynomial bounding the run time of M. For each x, let $Q(x)$ denote the unique query of M on x. Let W_0 (W_1) be the set of all x such that M on x rejects (accepts) no matter what the answer from the oracle is. Let R_0 (R_1) be the set of all x such that M on x accepts if and only if $Q(x)$ is not in the oracle ($Q(x)$ is in the oracle). Note that $W_0, W_1, R_0, R_1 \in$ P. Let $x_1, \ldots, x_m$ and n be such that $\max\{|x_1|, \ldots, |x_m|\} = n$ and $m \geq f(p(n))$. Define $h'(x_1, \ldots, x_m)$ as follows.

Case 1. $\{x_1, \ldots, x_m\} \cap (W_0 \cup W_1) \neq \emptyset$. Let j be the smallest i such that $x_i \in W_0 \cup W_1$. Define $h'(x_1, \ldots, x_m) = 0^{j-1}b0^{m-j}$, where $b = 1$ if $x_j \in W_0$ and 0 otherwise.

Case 2. $\{x_1, \ldots, x_m\} \cap (W_0 \cup W_1) = \emptyset$.

Subcase 2a. There is some $(i, j), i < j$ such that $Q(x_i) = Q(x_j)$. Let (k, l) be the smallest such pair. Define $h'(x_1, \ldots, x_m) = 0^{l-1}1^{n-l+1}$ if either $x_k, x_l \in R_0$ or $x_k, x_l \in R_1$, and $0^{k-1}1^{n-k+1}$ otherwise.

Subcase 2b. For every $i, j, i < j$, $Q(x_i) \neq Q(x_j)$. Let $b_1 \ldots b_m = h(x_1, \ldots, x_m)$. Define $h'(x_1, \ldots, x_m) = c_1 \ldots c_m$, where, for every i, $1 \leq i \leq m$, $c_i = b_i$ if $x_i \in R_1$ and $1 - b_i$ otherwise.

It is easy to observe that h' behaves correctly. □

Proposition 2.1 (2) states that P-mc(f) $\subseteq$ P-mc(g) if, for all but finitely many n, $f(n) \leq g(n)$. Interestingly, as we shall show below, P-mc(f) differs from P-mc(g) if, for infinitely many n, $f(n) \neq g(n)$. We note here that the proof we develop has a flavor similar to that of [5, Thm. 2].

THEOREM 2.2. *Let f and g be monotone nondecreasing, polynomial-time computable, polynomially bounded functions that map* $\mathbf{N}$ *to* $\mathbf{N}^+$. *Suppose for infinitely many n it holds that $g(n) > f(n)$. Then there is a set $A \in$* P-mc(g) $\setminus$ P-mc(f).

Proof. Let f and g be as in the hypothesis of the theorem. Since f is polynomially bounded, there is some n_0 such that for every $n \geq n_0$, $f(n) < 2^n$. Define a sequence $\{l_i\}_{i \geq 1}$ as follows:

(i) $l_1 = \min\{n \geq n_0 \mid g(n) > f(n)\}$;

(ii) for $i > 1$, $l_i = \min\{n \geq 2^{2^{l_{i-1}}} \mid g(n) > f(n)\}$.

By our hypothesis, $\{l_i\}_{i\geq 1}$ is a sequence of integers for which g is larger than f. Let $h_1, h_2, \ldots$ be an enumeration of all polynomial-time computable functions such that for each i, h_i is computable in time $n^i + i$.

We construct A in stages. At stage s we diagonalize against h_s by putting at most $f(l_s)$ strings of length l_s into A. Note that there are more than $f(l_s)$ strings of length l_s because $l_s > n_0$. The construction at stage s proceeds as follows: For each i, $1 \leq i \leq f(l_s)$, let w_i be the ith smallest string of length l_s. Let $b = h_s(w_1, \ldots, w_{f(l_s)})$. If $b \notin \{0,1\}^{f(l_s)}$, then h_s is already not an f-mc function. So, we proceed to the next stage, adding no new elements to A. If $b \in \{0,1\}^{f(l_s)}$, then for each i we put w_i into A if and only if the ith bit of b is a 1. This yields $h_s(w_1, \ldots, w_{f(l_s)}) = A(w_1)\ldots A(w_{f(l_s)})$, so h_s cannot be an f-mc function for A. Clearly, this construction establishes $A \notin \text{P-mc}(f)$.

Next we define a g-mc function γ for A. Let $y_1, \ldots, y_m$ be such that $m = g(\max\{|y_1|, \ldots, |y_m|\})$. Without loss of generality we may assume that $|y_1| \leq \cdots \leq |y_m|$. Let μ be the largest i such that $l_i \leq |y_m|$. There are the following four possible cases:

(a) For some i, $|y_i| \notin \{l_1, \ldots, l_\mu\}$. Clearly, for any such i, $y_i \notin A$. Define $\gamma(y_1, \ldots, y_m) = 1^m$.

(b) For some i, j, $i < j$, it holds that $y_i = y_j$. Clearly, for such i and j, $y_i \in A$ if and only if $y_j \in A$. So let s be the smallest i such that y_i appears in $y_{i+1}, \ldots, y_m$. Define $\gamma(y_1, \ldots, y_m) = 0^{s-1}1^{m-s+1}$.

(c) $|y_1| = \cdots = |y_m| = l_\mu$ and $y_1, \ldots, y_m$ are all distinct. By definition, it holds that $m > f(l_\mu)$. Since $A \cap \Sigma^{l_\mu}$ has at most $f(l_\mu)$ elements, some y_i is not in A. Define $\gamma(y_1, \ldots, y_m) = 1^m$.

(d) $|y_1|, \ldots, |y_m| \in \{l_1, \ldots, l_\mu\}$, $|y_m| = l_\mu$, $|y_1| = l_s$ for some $s < \mu$, and $y_1, \ldots, y_m$ are all distinct. We simulate the construction of A at stage s to compute $A^c(y_1)$ and define $\gamma(y_1, \ldots, y_m) = A^c(y_1)1^{m-1}$.

In each case, it holds that $\gamma(y_1, \ldots, y_m) \neq A(y_1)\ldots A(y_m)$. So, γ is a g-mc function for A.

It remains to show that γ is polynomial-time computable. Since f and g are both polynomial-time computable and $\{l_i\}_{i\geq 1}$ is a strictly increasing sequence, μ and $l_1, \ldots, l_\mu$ are computable in time polynomial in $|y_m|$. So, one can easily compute the value of γ for the cases (a), (b), and (c). Now suppose that case (d) holds. Since $s < \mu$, it holds that $s \leq |y_1| \leq \log\log|y_m|$. Since h_s is computable in time $n^s + s$ and f is polynomially bounded, for some fixed constant k the construction at stage s can be simulated in time $\mathcal{O}(s(|y_1|^k)^s) \leq \mathcal{O}(|y_1|^{2|y_1|^k}) \leq \mathcal{O}(2^{2^{|y_1|}}) \leq \mathcal{O}(|y_m|)$. Thus, the question of whether $y_1 \in A$ can be tested in time $\mathcal{O}(|y_m|)$. Therefore, γ is polynomial-time computable. This proves the theorem. □

COROLLARY 2.3. *If $f(n) > g(n)$ for infinitely many n and $g(n) > f(n)$ for infinitely many n, then by the above theorem,* P-mc(f) *and* P-mc(g) *are incomparable.*

COROLLARY 2.4. P-mc *classes construct a proper hierarchy according to the bound on the number of arguments; namely,* $\text{P} = \text{P-mc}(1) \subset \text{P-mc}(2) \subset \text{P-mc}(3) \subset \cdots \subset \text{P-mc}(k) \subset \text{P-mc}(k+1) \subset \cdots \subset \text{P-mc}(\text{const}) \subset \text{P-mc}(\log) \subset \text{P-mc}(\text{poly})$.

3. Relationships with P-selective sets. In this section we study relationships between polynomial-time membership comparable sets and P-selective sets. First of all, by definition, P-Sel is a subclass of P-mc(2).

PROPOSITION 3.1. $\text{P-Sel} \subseteq \text{P-mc}(2)$.

The above inclusion is proper.

THEOREM 3.2. *There is a tally set* $T \in \text{P-mc}(2) - \text{P-Sel}$.

Proof. Let $f_1, f_2, \ldots$ be an enumeration of all polynomial-time computable arity-2 functions. Let f_i be computable in time $p_i(n) = n^i + i$. Define $\mu(0) = 1$ and $\mu(n) = 2^{\mu(n-1)}$ for

$n > 0$, and $\nu(n) = \mu(4n)$. For each $i \geq 1$, let $u_i = 0^{\nu(i)}$. Our set T is constructed in stages. At stage i, we do the following:

(*) If $f_i(u_{2i}, u_{2i+1}) = u_{2i}$, then put u_{2i+1} into T. Otherwise, put u_{2i} into T.

Clearly, for any i either f_i is not a selector function or there exist some $x \in T$ and $y \in T^c$ such that either $f_i(x, y) = y$ or $f_i(y, x) = y$. So $T \notin$ P-Sel. We need to show that $T \in$ P-mc(2). Note that for any i, the construction up to stage i can be simulated in time $i \cdot p_i(2\nu(2i + 1)) < \nu(2(i + 1))$. Define a function g as follows: let $x = 0^s$ and $y = 0^t$ be tally strings.

Case 1. $s \neq \nu(k)$ for any k. Define $g(x, y) = 10$.

Case 2. $s = \nu(k)$ for some k and $t \neq \nu(l)$ for any l. Define $g(x, y) = 01$.

Case 3. $s = \nu(k)$ for some k and $t = \nu(l)$ for some l.

Subcase 3a. $\lfloor k/2 \rfloor < \lfloor l/2 \rfloor$. Simulate the construction of T up to stage $\lfloor k/2 \rfloor$ to test whether $x \in T$. Define $g(x, y) = 00$ if $x \in T$ and 10 otherwise.

Subcase 3b. $\lfloor k/2 \rfloor > \lfloor l/2 \rfloor$. Simulate the construction of T up to stage $\lfloor l/2 \rfloor$ to test whether $y \in T$. Define $g(x, y) = 00$ if $y \in T$ and 01 otherwise.

Subcase 3c. $\lfloor k/2 \rfloor = \lfloor l/2 \rfloor$. Define $g(x, y) = 11$ if $x \neq y$ and 01 otherwise.

It is easy to see that g witnesses the fact that $T \in$ P-mc(2). This proves the theorem. □

Are reducibility classes of P-Sel included in P-mc(poly)? The following theorem answers the question.

THEOREM 3.3. *Let $f : \mathbf{N} \to \mathbf{N}^+$ be a monotone nondecreasing function. Let L be $\leq^p_{f(n)\text{-}tt}$-reducible to a P-selective set. Then $L \in$ P-mc$((1 + c) \log f(n))$ for any constant $c > 0$.*

The proof of the theorem is based on Lemma 3.4 below of Toda, stating that, given a P-selective set A and a finite set Q, one can compute a linear order over Q such that $A \cap Q$ is the initial segment of the order. Originally, Jockusch [20] (attributed to Appel and McLaughlin) proved that being semi-recursive is equivalent to being the initial segment of a recursive linear ordering. Regarding P-selective sets, which are defined as the polynomial-time analogue of semi-recursive sets, Selman [32] showed that the initial segment of a polynomial-time linear order is a P-selective set. Ko [22] showed that being P-selective is equivalent to being the union of initial segments of polynomial-time preorder.

LEMMA 3.4 ([36]). *Let A be P-selective. There is a polynomial-time algorithm that, given a finite set $Q \subseteq \Sigma^*$, outputs an enumeration $y_1, \ldots, y_{\|Q\|}$ of elements in Q such that there exists some m, $0 \leq m \leq \|Q\|$, such that $A \cap Q = \{y_i \mid 1 \leq i \leq m\}$.*

Now we prove Theorem 3.3.

Proof of Theorem 3.3. Let f and L be as in the hypothesis. Let $L \leq^p_{f(n)\text{-}tt} A$ via a machine M and let A be P-selective. Let $c > 0$ and define $h(n) = \lfloor (1 + c) \log f(n) \rfloor$. Let n and $x_1, \ldots, x_{h(n)}$ be such that $n = \max\{|x_1|, \ldots, |x_m|\}$. For each i, $1 \leq i \leq h(n)$, let Q_i denote the set of all queries of M on x_i, and let $R = Q_1 \cup \cdots \cup Q_{h(n)}$. Since f is monotone nondecreasing, $\|Q_i\| \leq f(n)$ so, for sufficiently large n, it holds that

$$\|R\| \leq h(n) f(n) \leq 2^{\log h(n) + \log f(n)} \leq 2^{h(n)} - 2.$$

By Lemma 3.4, in time polynomial in $\sum_{y \in R} |y|$ and thus in time polynomial in $|x|$, we can compute an enumeration $y_1, \ldots, y_{\|R\|}$ of elements in R such that for some m, $0 \leq m \leq \|R\|$, $R \cap A = \{y_i \mid 1 \leq i \leq m\}$. Now for each m, $0 \leq m \leq \|R\|$, let $B_m = \{y_i \mid 1 \leq i \leq m\}$, and for each j, $1 \leq j \leq h(n)$, let $b_{m,j} = 1$ if M^{B_m} on x_j accepts and 0 otherwise. Clearly, there is some m such that for every j, $1 \leq j \leq h(n)$, $L(x_j) = b_{m,j}$. Since $\|R\| \leq 2^{h(n)} - 2$, there is some $v \in \{0, 1\}^{h(n)}$ such that $v \neq b_{m,1} \ldots b_{m,h(n)}$ for any m. Let v_0 be the smallest such v and define $r(x_1, \ldots, x_{h(n)}) = v_0$. It is easy to see that r witnesses the fact that $A \in$ P-mc(h). This proves the theorem. □

COROLLARY 3.5. $P_{btt}(\text{P-Sel}) \subseteq \text{P-mc(const)}$ *and* $P_{tt}(\text{P-Sel}) \subseteq \text{P-mc(log)}$.

A function h is said to be polynomially length bounded if there is a polynomial p such that for every x, $|h(x)| \leq p(|x|)$.

DEFINITION 3.6 ([21]). *A set L is in* P/poly *if there exist a polynomially length-bounded function h and a set $A \in \text{P}$ such that for every x, it holds that*

$$x \in L \leftrightarrow (x, h(0^{|x|})) \in A.$$

Ko [22] showed that P-selective sets have polynomial-size circuits. Noting for a P-selective set A, a finite set W, and a string x, that W is partitioned into two sets W_1, W_2, such that $x \in A \Longrightarrow W_1 \subseteq A$ and $x \in A^c \Longrightarrow W_2 \subseteq A^c$, Ko developed a divide-and-conquer method to find polynomially length-bounded advice. Such a method is, however, hard to find for P-mc(2) sets, because the set W is now partitioned into four sets $W_1, \ldots, W_4$ such that $x \in A \Longrightarrow W_1 \subseteq A, x \in A \Longrightarrow W_2 \subseteq A^c, x \in A^c \Longrightarrow W_3 \subseteq A^c$, and $x \in A^c \Longrightarrow W_4 \subseteq A^c$. Nonetheless, very surprisingly, P-mc(poly) sets have polynomial-size circuits, which is stated below.

THEOREM 3.7. P-mc(poly) $\subseteq$ P/poly.

The proof of the above theorem is essentially the same as that of [4, Thm. 10], so we omit the proof here. As a matter of fact, in [4], Amir, Beigel, and Gasarch showed that, for any $k \geq 1$, P-mc(k) $\subseteq$ P/poly, developing an algorithm to construct an advice string of length $\mathcal{O}(kn^2)$ for Σ^n. Thus, even if k is a function of n that is polynomially bounded, their construction still works.

It is well known that P/poly $= R_{tt}(\text{TALLY}) = R_{tt}(\text{SPARSE})$ and TALLY $\subseteq R_T(\text{P-Sel})$ [30]. So every set in P-mc(poly) is $\leq_T^p$-reducible to some P-selective set.

COROLLARY 3.8. P-mc(poly) $\subseteq R_T(\text{P-Sel})$.

Since $R_{tt}(\text{P-Sel}) \subseteq \text{P-mc(log)}$ and P-mc(log) $\neq$ P-mc(poly), the above inclusion is optimal.

COROLLARY 3.9. P-mc(poly) $\not\subseteq R_{tt}(\text{P-Sel})$.

The converse of Theorem 3.7 does not appear to hold. The question of whether the converse holds is very subtle, because proving P-mc(poly) $\neq$ P/poly is at least as hard as proving P $\neq$ NP.

THEOREM 3.10. *If* P = NP, *then* P/poly $\subseteq$ P-mc(poly).

Proof. Suppose P = NP. Let $L \in$ P/poly. Since P/poly $= R_{tt}(\text{TALLY})$, there is a tally set T and a polynomial time-bounded deterministic oracle Turing machine M such that for every x, $x \in L$ if and only if M^T on x accepts. Let p be a polynomial bounding the run time of M. Without loss of generality we may assume for every x and oracle X, that any query of M^X on x is in $\{0, \ldots, 0^{p(|x|)}\}$. Let $x_1, \ldots, x_m$ be strings of length at most n with $m = p(n) + 1$. Let $T_1, \ldots, T_k$ be an enumeration of all subsets of $\{0^1, \ldots, 0^{p(n)}\}$, where $k = 2^{p(n)}$. Note that every T_i can be represented by a string of length $p(n)$. There is some i, $1 \leq i \leq k$, such that for every j, $1 \leq j \leq m$, M^T on x_j accepts if and only if M^{T_i} on x_j accepts. For each i, $1 \leq i \leq k$, and j, $1 \leq j \leq m$, let $b(i, j) = 1$ if M^{T_i} on x_j accepts and 0 otherwise. For each i, $1 \leq i \leq k$, let $c_i = (b(i, 1), \ldots, b(i, m))$. Clearly, there is some i, $1 \leq i \leq k$, such that $c_i = (L(x_1), \ldots, L(x_m))$. Since $m = p(n) + 1$, there is some $c \in \{0, 1\}^m$ such that $c \neq c_i$ for any i. Let $\hat{c}$ be the smallest such c and define $g(x_1, \ldots, x_m) = \hat{c}$. Then, $\hat{c} \neq (L(x_1), \ldots, L(x_m))$. By our supposition that P = NP, as a representation of T_i ranges over strings of length $p(n)$, the above $\hat{c}$ can be computed in time polynomial in n, so g is polynomial-time computable. Therefore, $L \in \text{P-mc}(p + 1)$. Hence, P/poly $\subseteq$ P-mc(poly). □

We conjecture that P-mc(poly) is a proper subclass of P/poly.

4. Relationships with sparse and tally sets. In this section we study relationships between polynomial-time membership comparable sets and sparse sets as well as tally sets. Since P/poly $= R_{tt}(\text{SPARSE}) = R_{tt}(\text{TALLY})$, by Theorem 3.7 it holds that P-mc(poly) $\subseteq R_{tt}(\text{SPARSE})$ and P-mc(poly) $\subseteq R_{tt}(\text{TALLY})$. Moreover, as P/poly $= R_T(\text{P-Sel})$ and P-Sel $\subseteq$ P-mc(2), we have SPARSE $\subseteq R_T(\text{P-mc(poly)})$. But, in fact, it holds that SPARSE $\subseteq$ P-mc(poly).

PROPOSITION 4.1. SPARSE $\subseteq$ P-mc(poly).

Proof. Let S be a sparse set. There is a polynomial p such that for every n, $\|S \cap \Sigma^{\leq n}\| < p(n)$. Define g as a function that, given $x_1, \ldots, x_m$ with $m \geq p(n)$ and $n = \max\{|x_1|, \ldots, |x_m|\}$, outputs 1^m. The function g is polynomial-time computable. For every $x_1, \ldots, x_m$ with $m \geq p(n)$ and $n = \max\{|x_1|, \ldots, |x_m|\}$, it cannot happen that $x_1, \ldots, x_m \in S$ because $\|S \cap \Sigma^{\leq n}\| < p(n)$. Thus, g witnesses the fact that $S \in$ P-mc(poly). □

Thus, it is possible for an uncountable number of sets to be in P-mc(poly) via the same function g.

By Proposition 2.1 (3), we have the following theorem.

THEOREM 4.2. $R_{1\text{-}tt}(\text{SPARSE}) \subseteq$ P-mc(poly).

For any tally set T, let $r(T)$ denote $T(0)T(00)T(000)\ldots$. The string $r(T)$ can be viewed as a binary real number. For a binary real number $r(T) \in [0, 1)$, define Left-Cut[r] as the set of binary strings w smaller than or equal to $r(T)$ in the dictionary order and Prefix[r] as the set of all initial bits of $r(T)$. Selman [30] showed the following theorem.

THEOREM 4.3. *For any binary real number $r(T) \in [0, 1)$, the following properties hold:*

1. Left-Cut[r] *is P-selective.*
2. Left-Cut[r] $\leq^p_{ptt} T$ *and* $T \leq^p_T$ Left-Cut[r].
3. *If* Prefix[r] *is P-selective, then* $T \in$ P.
4. Prefix[r] $\leq^p_{tt} T$ *and* $T \leq^p_T$ Prefix[r].

How complex is Prefix[r]? We show below that Prefix[r] is polynomial-time two-membership comparable.

THEOREM 4.4. *For any* $r \in [0, 1)$, Prefix[r] *is in* P-mc(2).

Proof. Let $r \in [0, 1)$. Let x, y be distinct two strings. Then the following properties hold:

(i) If x is a prefix of y, then $y \in$ Prefix[r] $\rightarrow x \in$ Prefix[r].

(ii) If y is a prefix of x, then $x \in$ Prefix[r] $\rightarrow y \in$ Prefix[r].

(iii) If x is not a prefix of y and y is not a prefix of x, then at most one of x and y is in Prefix[r].

Define $g(x, y) = 01$ if the first condition is satisfied, 10 if the second condition is satisfied, and 11 otherwise. Clearly, g witnesses the fact that Prefix[r] $\in$ P-mc(2). □

COROLLARY 4.5. *For any tally set T, there is a sparse set $S \in$ P-mc(2) such that $T \leq^p_T S$ and $S \leq^p_{tt} T$.*

Since there is a tally set not in P, we have the following corollary.

COROLLARY 4.6. *There is a tally set T such that* Prefix[$r(T)$] *is in* P-mc(2) − P-Sel.

It is well known that for every set $A \in$ P/poly, there is a tally set $T \in \Delta^p_3(A)$ such that $A \leq^p_{tt} T$ (see, for example, [29]). By Proposition 4.1, TALLY $\subseteq$ P-mc(poly). So we have the following corollary.

COROLLARY 4.7. *For every $A \in$ P-mc(poly), there is a tally set $T \in$ P-mc(poly) $\cap \Delta^p_3(A)$ such that $A \leq^p_{tt} T$.*

Moreover, by Corollary 4.5, for every tally set T there is a sparse set $S \in$ P-mc(2) such that T and S are $\leq^p_T$-equivalent. Therefore, we have the following corollary.

COROLLARY 4.8. *For every $A \in$ P-mc(poly), there is a sparse set S in* P-mc(2) $\cap \Delta^p_3(A)$ *such that $A \leq^p_T S$.*

As SPARSE $\subseteq$ P-mc(poly), we have the following corollary.

COROLLARY 4.9. *For every sparse set S, there is a sparse set $S' \in \text{P-mc}(2) \cap \Delta_3^p(S)$ such that $S \leq_T^p S'$. Therefore,*

$$\text{P/poly} = R_{tt}(\text{SPARSE}) = R_T(\text{SPARSE} \cap \text{P-mc}(2)).$$

5. Polynomial-time membership comparable hard sets. In this section, we show for some complexity classes $\mathcal{C}$ that if, for some $c < 1$, $\mathcal{C} \subseteq \text{P-mc}(c \log n)$, then $\mathcal{C} = \text{P}$. We note here that the author has been recently informed that some of the results in this section had been independently proven by Agrawal and Arvind [1] and Beigel, Kummer, and Stephan [9]. We start by considering NP.

THEOREM 5.1. *If* $\text{NP} \subseteq \text{P-mc}(c \log n)$ *for some* $c < 1$, *then* $\text{P} = \text{NP}$.

Proof. Suppose that $\text{NP} \subseteq \text{P-mc}(c \log n)$ for some $c < 1$. Take a as a natural number such that $c < 1 - 1/a$. Consider an NP-complete set SAT. Without loss of generality we may assume that

(*) for every formula φ, each truth assignment for φ is encoded into a string of length $|\varphi|^{1/2a}$.

Let φ be a formula and Y be a set of prefixes of truth assignments for φ. Call Y *good* for φ if Y contains a prefix of a satisfying assignment for φ. By our assumption, there is an encoding $\varphi\#Y$ of φ and Y such that if $\|Y\| \leq |\varphi| - 1$, then $|\varphi\#Y| = |\varphi|^{1+1/a}$. Define $A = \{\varphi\#Y \mid Y \text{ is good for } \varphi\}$. Obviously, $A \in \text{NP}$, so $A \in \text{P-mc}(c \log n)$. Let f be a membership comparing function witnessing this property.

Let φ be a formula with $|\varphi| = r = 2^d$ and $Z = \{y_1, \ldots, y_{r-1}\}$ be a set of $r - 1$ many prefixes of truth assignments for φ. Suppose that the sets of truth assignments represented by these prefixes are disjoint; that is, for any i, j, $1 \leq i < j \leq r - 1$, y_i is not a prefix of y_j and y_j is not a prefix of y_i. For each i, $1 \leq i \leq d$, let Y_i denote the set of all y_j, $1 \leq j \leq r - 1$, such that the ith bit of j's binary representation in $\{0, 1\}^d$ is a 1; that is, j's representation is of the form $b_1 \ldots b_d$ with $b_i = 1$. Note for any i, $1 \leq i \leq d$, that $\|Y_i\| = 2^{d-1} = r/2$, so $|\varphi\#Y_i| = |\varphi|^{1+1/a} = r^{1+1/a} = 2^{d(1+1/a)}$, and thus $c \log |\varphi\#Y_i| < cd(1 + 1/a) < (1 - 1/a^2)d < d$. So, given $\varphi\#Y_1, \ldots, \varphi\#Y_d$ as arguments, f must exclude one possibility of $(A(\varphi\#Y_1), \ldots, A(\varphi\#Y_d))$; that is, $f(\varphi\#Y_1, \ldots, \varphi\#Y_d)$ maps to some $b = b_1 \ldots b_d \in \{0, 1\}^d$ so that $(A(\varphi\#Y_1), \ldots, A(\varphi\#Y_d)) \neq b$.

Suppose that $b = 0^d$. Then, $(A(\varphi\#Y_1), \ldots, A(\varphi\#Y_d)) \neq 0^d$ holds. So, at least one of $\varphi\#Y_1, \ldots, \varphi\#Y_d$ is in A, and thus at least one of $Y_1, \ldots, Y_d$ is good for φ. Therefore φ is satisfiable.

On the other hand, suppose that $b \in \{0, 1\}^d - \{0^d\}$. Let t be the number whose binary representation is $b = b_1 \ldots b_d$. We show that if Z is good then $Z - \{y_t\}$ is good. Assume, by way of contradiction, that Z is good but $Z - \{y_t\}$ is not good. Then for every i, $\varphi\#Y_i \in A$ if and only if $y_t \in Y_i$. On the other hand, for every i, $y_t \in Y_i$ if and only if the ith bit of t, which is b_i, is a 1. Therefore, for every i, $\varphi\#Y_i \in A$ if and only if $b_i = 1$. This implies $(A(\varphi\#Y_1), \ldots, A(\varphi\#Y_d)) = b_1 \ldots b_d = b$, which contradicts f's declaration that $(A(\varphi\#Y_1), \ldots, A(\varphi\#Y_d)) \neq b$. So, if Z is good then $Z - \{y_t\}$ is good. Moreover, if $Z - \{y_t\}$ is good then, since it is a subset of Z, Z is good too. Therefore, in this case it holds that Z is good if and only if $Z - \{y_t\}$ is good.

Define PRUNE as a procedure that, given φ and Z as above, (1) computes $b = f(\varphi\#Y_1, \ldots, \varphi\#Y_d)$ and (2) outputs YES if b is all 0 and outputs $Z - \{y_t\}$ otherwise. Clearly, PRUNE is a polynomial-time procedure and for every φ and Z, it holds that

(i) if PRUNE outputs YES, then $\varphi \in \text{SAT}$, and

(ii) if PRUNE outputs a set Z', then Z' is good if and only if Z is good.

Moreover, when Z consists of truth assignments for φ, the question of whether Z is good can be tested in time polynomial in $|\varphi|$.

Now consider a decision procedure that, given a formula φ, behaves as follows:

(0) Initially, set Z to $\{\lambda\}$.

(1) According to $\|Z\|$, do the following:

(1a) $\|Z\| < |\varphi| - 1$. Let z be the smallest $y \in Z$ in canonical lexicographic order and replace z with $z0$ and $z1$.

(1b) $\|Z\| = |\varphi| - 1$. Call PRUNE(φ, Z). If PRUNE outputs YES, then accept φ. Otherwise, replace Z with the output of PRUNE.

(2) If not all $y \in Z$ are truth assignments for φ, then goto (1). Otherwise, accept φ if and only if Z is good.

It is not hard to see that the procedure is polynomial-time bounded and accepts φ if and only if φ is satisfiable. Therefore SAT $\in$ P. This proves the theorem. □

Remark 1. We note here that the above proof does not seem to work for the case $c \geq 1$, even if we wish to prove a consequence weaker than P = NP. Let φ be of length 2^d. Suppose we wish to preserve $\|Z\| \leq 2^{H(d)} - 1$ for some function H. (Note that $H(d) = \mathcal{O}(d)$ if we wish to develop a polynomial-time algorithm.) In order to eliminate one prefix from Z, we construct subsets $Y_1, \ldots, Y_{H(d)}$ of Z, each consisting of $2^{H(d)-1}$ prefixes. Let t be the average length of prefixes in Z. Then, $\varphi\#Y_i$'s must encode at least $2^d + 2^{H(d)-1}t$ bits in average. Thus, for some i, $|\varphi\#Y_i| \geq \alpha(2^d + 2^{H(d)-1}t)$ holds, where α is a constant depending only on the size of the encoding alphabet. Now, the number of arguments we must give to f is at least

$$\begin{aligned} c \log \max\{|\varphi\#Y_1|, \ldots, |\varphi\#Y_{H(d)}|\} &\geq c \log(\alpha(2^d + 2^{H(d)-1}t)) \\ &> c \log \alpha + c \log(2^d + 2^{H(d)-1}t) \\ &> cH(d) + c \log(2^{d-H(d)} + t/2). \end{aligned}$$

Because t cannot be bounded by any constant, we may assume $\log(2^{d-H(d)} + t/2) \geq 1$. So, we need more than $cH(d) + c$ arguments, but if $c \geq 1$ this is impossible, because we have only $H(d)$ arguments. The same arguments apply for the proof of Theorem 5.5.

Next we consider subclasses of NP; namely, UP and FewP, which are defined by Valiant [37] and Allender [2], respectively. For a polynomial time-bounded nondeterministic Turing machine M, let $\#\mathrm{acc}_M$ denote the function that maps x to the number of accepting computation paths of M on input x. A set L is in UP (respectively, FewP) if there is a polynomial time-bounded nondeterministic Turing machine M witnessing $L \in$ NP such that, for every x, $\#\mathrm{acc}_M(x) \leq 1$ (respectively, for every x, $\#\mathrm{acc}_M(x) \leq p(|x|)$ for some polynomial depending only on M). By Cook's reduction [14] and padding arguments (see, for example, [10]), for any NP-acceptor M and any $a \in \mathbf{N}^+$, one can construct a polynomial-time computable function f such that the following conditions are satisfied:

(i) $f(\Sigma^*)$ is a set of formulas, and is in P;

(ii) for every x, $\#\mathrm{acc}_M(x)$ equals the number of satisfying assignments for $f(x)$;

(iii) for every x, truth assignments for $f(x)$ are of length $|f(x)|^{1/2a}$.

Define $S = f(\Sigma^*) \cap \mathrm{SAT}$ and define A as in the proof of Theorem 5.1 with S in place of SAT. Then $L(M) \leq_m^p S$, $S \leq_m^p A$, and $S, A \in$ UP if $L(M) \in$ UP ($S, A \in$ FewP if $L(M) \in$ FewP). Thus, we can use our technique to prove results similar to that of Theorem 5.1 for UP and FewP.

THEOREM 5.2. *If* UP $\subseteq$ P-mc($c \log n$) *for some* $c < 1$, *then* UP = P. *Therefore, if* UP $\subseteq R_{n^c\text{-}tt}$(P-Sel) *for some* $c < 1$, *then* UP = P.

THEOREM 5.3. *If* FewP $\subseteq$ P-mc($c \log n$) *for some* $c < 1$, *then* FewP = P. *Therefore, if* UP $\subseteq R_{n^c\text{-}tt}$(P-Sel) *for some* $c < 1$, *then* FewP = P.

Next we consider counting complexity classes $\mathrm{C}_=\mathrm{P}$ and PP, and PSPACE. A set A is in PP [15], [33] ($\mathrm{C}_=\mathrm{P}$ [38], [33]) if there exist some machines M and N such that for every x, $x \in A$ if and only if $\#\mathrm{acc}_M(x) \geq \#\mathrm{acc}_N(x)$ ($\#\mathrm{acc}_M(x) = \#\mathrm{acc}_N(x)$).

THEOREM 5.4. *Let $\mathcal{C}$ be in* {PP, $C_=P$, PSPACE}. *If* $\mathcal{C} \subseteq$ P-mc($c \log n$) *for some* $c < 1$, *then* $\mathcal{C} =$ P.

Proof. Note that NP is a subset of either $\mathcal{C}$ or co-$\mathcal{C}$. Since $\mathcal{C}$ and co-$\mathcal{C}$ are both contained in P-mc($c \log n$), by Theorem 5.1, P = NP. Hence P = Σ_2^p. Since each of $C_=P$, PP, and PSPACE has "one word-decreasing self-reducible" $\leq_m^p$-complete sets [28] and since, if a "word-decreasing self-reducible" set A is in P/poly, then $\Sigma_2^p(A) \subseteq \Sigma_2^p$ [21], we have $\mathcal{C} \subseteq \Sigma_2^p$. This establishes the fact that $\mathcal{C} =$ P. □

Let $k \geq 2$. A set A is in MOD_kP [13] if there is some machine M such that for every x, $x \in A$ if and only if $\#acc_M(x) \not\equiv 0$ modulo k. The argument for $C_=P$, PP, and PSPACE cannot be applied to MOD_kP, because it is not known whether NP or coNP is included in MOD_kP. So, we need to develop a direct proof.

THEOREM 5.5. *Let* $k \geq 2$. *If* $MOD_kP \subseteq$ P-mc($c \log n$) *for some* $c < 1$, *then* $MOD_kP =$ P.

Proof. Let $k \geq 2$ and suppose $MOD_kP \subseteq$ P-mc($c \log n$) for some $c < 1$. Let a be a natural number such that $c < 1 - 1/a$. Without loss of generality, we may assume for any formula φ of propositional logic and any truth assignment y for φ that $|y| = |\varphi|^{1/2a}$. For a formula φ and y, $|y| \leq |\varphi|^{1/2a}$, let

$$\mu(\varphi, y) = \|\{yz : yz \text{ is a satisfying assignment for } \varphi\}\| \text{ modulo } k.$$

Note that for every φ and y, $\mu(\varphi, y) \in \{0, \ldots, k-1\}$. Define $L_0 = \{\varphi : \mu(\varphi, \lambda) \neq 0\}$ and $L_1 = \{(\varphi, l) : \mu(\varphi, \lambda) = l\}$. It is well known that L_0 is $\leq_m^p$-complete for MOD_kP and it is clear that L_0 is $\leq_{(k-1)\text{-}dtt}^p$-reducible to L_1. We will show that L_1 is in P. Let φ be a formula and Y be a set of pairs $(y_1, m_1), \ldots, (y_d, m_d)$ such that for all i, $1 \leq i \leq d$, $|y_i| \leq |\varphi|^{1/2a}$ and $m_i \in \{0, \ldots, k-1\}$. Call Y *good* for φ if, for every i, $1 \leq i \leq d$, $\mu(x, y_i) = m_i$. By our assumption on the length of formulas and their truth assignments, there is an encoding $\varphi\#Y$ such that if $\|Y\| \leq |\varphi|/2$, then $|\varphi\#Y| = |\varphi|^{1+1/a}$. Define A as the set of all $\varphi\#Y$ such that Y is good for φ. Since MOD_kP is closed under $\leq_{dtt}^p$-reductions [19], and thus co-MOD_kP is closed under $\leq_{ctt}^p$-reductions by symmetry, we have $A \in$ co-MOD_kP. So, by our supposition, A^c is in P-mc($c \log n$). Let f be a function witnessing this property.

We will show that $L_1 \in$ UP. Let φ be a formula such that $|\varphi| = r = 2^d$ and let $Z = \{(y_i, z_i) : 1 \leq i \leq r-1\}$ be such that for every i, $1 \leq i \leq r-1$, $|y_i| \leq r^{1/2a}$ and $m_i \in \{0, \ldots, k-1\}$. Moreover, suppose for every i, j, $1 \leq i < j \leq r-1$, that y_i is not a prefix of y_j and y_j is not a prefix of y_i. For each l, $1 \leq l \leq d$, let Y_l be the set of all (y_i, z_i) such that the lth bit of i's binary representation in $\{0, 1\}^d$ is a 1; that is, i's representation is of the form $b_1 \ldots b_d$ with $b_l = 1$. It is easy to see that each Y_l contains exactly $r/2$ pairs, and thus that $|\varphi\#Y_l| = r^{1+1/a} = 2^{d(1+1/a)}$. Let $b = f(\varphi\#Y_1, \ldots, \varphi\#Y_d)$. Since $c \log |\varphi\#Y_l| \leq cd(1 + 1/a) < d(1 - 1/a^2) < d$, b must be of length d and differ from $(A^c(\varphi\#Y_1), \ldots, A^c(\varphi\#Y_d))$.

Suppose that $b \in 0^d$. Then there is some l such that $\varphi\#Y_l \in A^c$, so Y_l is not good for φ, and thus Z is not good for φ because each Y_l is a subset of Z.

On the other hand, suppose that $b \in \{0, 1\}^d - \{0^d\}$. For each i, let b_i denote the ith bit of b. Let t be the number whose binary representation is b. We show that Z is good if $Z - \{(y_t, m_t)\}$ is good. Assume, by way of contradiction, that $Z - \{(y_t, m_t)\}$ is good and Z is not good. By definition, for every l, Y_l is good if and only if $(y_t, m_t) \notin Y_l$. On the other hand, for every l, $(y_t, m_t) \in Y_l$ if and only if the lth bit of t's binary representation, which is b_l, is a 1. So, for every l, Y_l is good if and only if $b_l = 0$. Thus, $(A^c(\varphi\#Y_1), \ldots, A^c(\varphi\#Y_d)) = b_1 \ldots b_d = b$, which contradicts f's declaration that $(A^c(\varphi\#Y_1), \ldots, A^c(\varphi\#Y_d)) \neq b$. Therefore, Z is good if $Z - \{(y_t, m_t)\}$ is good. Moreover, if Z is good then, obviously, for any nonempty subset Y of Z, Y is good. Hence, Z is good if and only if $Z - \{(y_t, m_t)\}$ is good.

Now define PRUNE as a procedure that, given x and Z as above, computes $b = f(\varphi\#Y_1, \ldots, \varphi\#Y_d)$ and outputs NO if the value is all 0 and $Z' = Z - \{(y_l, m_l)\}$ otherwise. Then the following properties hold:

(i) PRUNE is a polynomial-time procedure;

(ii) if PRUNE outputs NO, then Z is not good;

(iii) if PRUNE outputs Z', then Z is good if and only if Z' is good.

Moreover, if Z consists only of pairs of the form (y, m) with y being a truth assignment for φ, then the question of whether Z is good can be easily tested, because Z is good if and only if, for every $(y, m) \in Z$, it holds that $m = 1$ if y is a satisfying assignment for x and $m = 0$ otherwise.

Now define M as a nondeterministic Turing machine that, on input (φ, l), behaves as follows:

(0) Initially, set Z to $\{(\lambda, l)\}$.

(1) According to $\|Z\|$, do one of the following:

(1a) $\|Z\| = |\varphi| - 1$. Call PRUNE(φ, Z). If PRUNE outputs NO, then reject and halt. Otherwise, set Z to the output of PRUNE.

(1b) $\|Z\| < |\varphi| - 1$. Let $Z = \{(y_i, m_i) : 1 \leq i \leq d\}$ and let y_l be the smallest in $\{y_1, \ldots, y_d\}$ in canonical lexicographic order. Nondeterministically guess $n_0, n_1 \in \{0, \ldots, k-1\}$ such that $n_0 + n_1 \equiv m_r$ modulo k and replace (y_r, m_r) with two elements $(y_r 0, n_0)$ and $(y_r 1, n_1)$.

(2) If there is some (y, m) in Z such that y is not a truth assignment for φ, then goto (1). Otherwise, accept φ if and only if Z is good.

Suppose that M on input (φ, l) is at the start of step (1) with Z. Suppose that W is good. If M is to enter (1a), then Z is replaced with a good one, and if M is to enter (1b), then, clearly, there uniquely exists a guess of (n_0, n_1), for which Z is substituted with a good one. On the other hand, suppose that Z is not good. If M is to enter (1a), then M either rejects or substitutes Z with one that is not good, and if M is to enter (1b), then for every guess of (n_0, n_1), M substitutes Z with one that is not good. So, if (φ, l) is in L_1, then there exists a unique path leading to step (2) with a good Z; and if $(\varphi, l) \notin L_1$, there is no such path. Therefore, if $(\varphi, l) \in L_1$, there uniquely exists a path leading to acceptance, and if $(\varphi, l) \notin L_1$, then there exist no such paths. This implies $L_1 \in$ UP. So $\mathrm{MOD}_k\mathrm{P} \subseteq$ UP. Since UP $\subseteq \mathrm{MOD}_k\mathrm{P}$, by Theorem 5.2 we have UP = P. Hence $\mathrm{MOD}_k\mathrm{P} = \mathrm{P}$. This proves the theorem. □

The proof techniques we have developed enable us to resolve some open questions. Selman [30] showed if NP $\subseteq R_m$(P-Sel), then P = NP. It has been studied whether a similar statement holds for more flexible reducibilities. But, it has been open for a long time whether NP $\subseteq R_{btt}$(P-Sel) implies P = NP. By Theorems 3.3 and 5.1, we give an affirmative answer to this question.

COROLLARY 5.6. NP $\subseteq R_{btt}$(P-Sel) *implies* P = NP. *In fact,* NP $\subseteq R_{n^c\text{-}tt}$(P-Sel) *implies* P = NP *for any* $c < 1$.

Theorem 5.1 yields another consequence. For a set A and a function $f : \mathbf{N} \to \mathbf{N}$, let $\mathrm{FP}^A_{f(n)\text{-}T}$ ($\mathrm{FP}^A_{f(n)\text{-}tt}$) denote the class of functions that are polynomial-time computable with at most $f(|x|)$ adaptive (nonadaptive) queries to A. Krentel [23] showed that for any $f(n) \leq \frac{1}{2}\log n$, $\mathrm{FP}^{\mathrm{SAT}}_{f(n)\text{-}T} \subseteq \mathrm{FP}^{\mathrm{SAT}}_{(f(n)-1)\text{-}T}$ if and only if P = NP. Krentel asked whether a similar result holds for a larger function f. Beigel [6] strengthened the bound to $c \log n$ for any constant $c < 1$. He further asked a similar question with $\mathrm{FP}^{\mathrm{SAT}}_{f(n)\text{-}tt}$ in place of $\mathrm{FP}^{\mathrm{SAT}}_{f(n)\text{-}T}$ and showed that for any $\leq^p_{1\text{-}tt}$-hard set A for NP, any constant $c < 1$, and any f such that $f(n) \leq c \log n$, if $\mathrm{FP}^A_{f(n)\text{-}tt} \subseteq \mathrm{FP}^X_{(f(n)-1)\text{-}T}$ for some X, then RP = NP and P = UP. We prove that the conclusion of Beigel's result can be strengthened to P = NP.

THEOREM 5.7. *Let* $f(n) \leq c \log n$ *for some constant* $c < 1$. *Let* B *be* $\leq^p_{1\text{-}tt}$*-hard for* NP. *If, for some set* X, *it holds that* $\mathrm{FP}^B_{f(n)\text{-}tt} \subseteq \mathrm{FP}^X_{(f(n)-1)\text{-}T}$, *then* P = NP.

Proof. The proof is quite similar to that of Theorem 5.1. Let f, c, B, and X be as in the hypothesis. Without loss of generality we may assume that $f(n) = \lfloor (1 - 1/a) \log n \rfloor$ for some natural number a. Define the notion of "good" sets, the encoding $x\#Z$, and the set A as in the proof of Theorem 5.1. For every φ and a set Z of at most $|\varphi| - 1$ prefixes of truth assignments for φ, $|\varphi\#Z| = |\varphi|^{1+1/a}$. Define $g(n) = f(n^{1+1/a})$. Then, $g(n) < \log n$ for all n. Let h be a function that, given φ and a set Z of at most $|\varphi| - 1$ prefixes of truth assignments for φ, outputs $(A(\varphi\#Y_1), \ldots, A(\varphi\#Y_{g(|\varphi|)}))$, where Y_i's are subsets of Z defined in the proof of Theorem 5.1. Since $|\varphi\#Z| = |\varphi|^{1+1/a}$, $h \in \mathrm{FP}^{A}_{f(n)\text{-}tt}$, and thus $h \in \mathrm{FP}^{B}_{f(n)\text{-}tt}$. So, by our supposition, $h \in \mathrm{FP}^{X}_{(f(n)-1)\text{-}T}$. Let M be a machine witnessing the fact that $h \in \mathrm{FP}^{X}_{(f(n)-1)\text{-}T}$. For every φ and Z, there are $2^{g(|\varphi|)-1}$ possible outputs of M. Since $g(n) < \log n$, all such values can be computed in time polynomial in $|\varphi|$. Moreover, since there are $2^{g(|\varphi|)}$ possible values of $(A(\varphi\#Y_1), \ldots, A(\varphi\#Y_{g(|\varphi|)}))$, we can compute, in time polynomial in $|\varphi|$, a value $v \in \{0,1\}^{g(|\varphi|)}$, which is not equal to $(A(\varphi\#Y_1), \ldots, A(\varphi\#Y_{g(|\varphi|)}))$. Now let h' be a function that, given $\varphi\#Z$, maps to $v0^{\log|\varphi| - g(|\varphi|)}$. Clearly, h' is polynomial-time computable, and $(A(\varphi\#Y_1), \ldots, A(\varphi\#Y_{\log|\varphi|})) \neq h'(\varphi\#Z)$.

Therefore, as in the proof of Theorem 5.1, we can define a polynomial-time decision procedure for SAT. This proves the theorem. □

The above two results can be applied to other complexity classes.

COROLLARY 5.8. *Let $\mathcal{C}$ be a class chosen from* { PSPACE, UP, FewP, $\mathrm{C_{=}P}$, PP, $\mathrm{MOD_2P}$, $\mathrm{MOD_3P}$, ...}.

1. *If $\mathcal{C} \subseteq R_{n^c\text{-}tt}(\text{P-Sel})$ for some $c < 1$, then $\mathcal{C} = \mathrm{P}$.*
2. *Let H be $\leq^{p}_{1\text{-}tt}$-hard for A and let $f(n) \leq c \log n$ for some $c < 1$. If $\mathrm{FP}^{H}_{f(n)\text{-}tt} \subseteq \mathrm{FP}^{X}_{(f(n)-1)\text{-}T}$ for some X, then $\mathcal{C} = \mathrm{P}$.*

Acknowledgment. The author thanks Lane Hemaspaandra for useful discussions and anonymous referees for invaluable comments.

REFERENCES

[1] M. AGRAWAL AND V. ARVIND, *Polynomial time truth-table reductions to P-selective sets*, in Proc. 9th Conference on Structure in Complexity Theory, IEEE Computer Society Press, 1994, pp. 24–30.

[2] E. ALLENDER, *Invertible functions*, Ph.D. thesis, Department of Computer Science, Georgia Institute of Technology, 1985.

[3] E. ALLENDER AND L. HEMACHANDRA, *Lower bounds for the low hierarchy*, J. Assoc. Comput. Mach., 39 (1992), pp. 234–251.

[4] A. AMIR, R. BEIGEL, AND W. GASARCH, *Some connections between bounded query classes and non-uniform complexity*, in Proc. 5th Conference on Structure in Complexity Theory, IEEE Computer Society Press, 1990, pp. 232–243.

[5] A. AMIR AND W. GASARCH, *Polynomial terse sets*, Inform. and Comput., 77 (1988), pp. 37–56.

[6] R. BEIGEL, *NP-hard sets are P-superterse unless R=NP*, Tech. report 88-04, Department of Computer Science, The Johns Hopkins University, Baltimore, MD, 1988.

[7] ———, *Bounded queries to SAT and the Boolean hierarchy*, Theoret. Comput. Sci., 84 (1991), pp. 199–223.

[8] R. BEIGEL, W. GASARCH, J. GILL, AND J. OWINGS, *Terse, superterse, and verbose sets*, Inform. and Comput., 103 (1988), pp. 68–85.

[9] R. BEIGEL, M. KUMMER AND F. STEPHAN, *Approximable sets*, in Proc. 9th Conference on Structure in Complexity Theory, IEEE Computer Society Press, 1994, pp. 12–23.

[10] L. BERMAN AND J. HARTMANIS, *On isomorphisms and density of NP and other complete sets*, SIAM J. Comput., 6 (1977), pp. 305–322.

[11] H. BUHRMAN, P. VAN HELDEN, AND L. TORENVLIET, *P-selective self-reducible sets: A new characterization of P*, in Proc. 8th Conference on Structure in Complexity Theory, IEEE Computer Society Press, 1993, pp. 44–51.

[12] J. CAI AND L. HEMACHANDRA, *Enumerative counting is hard*, Inform. and Comput., 82 (1989), pp. 34–44.

[13] ———, *On the power of parity polynomial time*, Math. Systems Theory, 23 (1990), pp. 95–106.

[14] S. COOK, *The complexity of theorem proving procedures*, in Proc. 3rd Symposium on Theory of Computing, ACM Press, 1971, pp. 151–158.

[15] J. GILL, *Computational complexity of probabilistic Turing machines*, SIAM J. Comput., 6 (1977), pp. 675–695.
[16] L. HEMACHANDRA, A. HOENE, M. OGIWARA, A. SELMAN, T. THIERAUF, AND J. WANG, *Selectivity*, in Proc. 5th International Conference on Computing and Information, IEEE Computer Society Press, 1993, pp. 55–59.
[17] E. HEMASPAANDRA, A. NAIK, M. OGIWARA, AND A. SELMAN, *P-selective sets, and reducing search to decision vs. self-reducibility*, Tech. Report TR 93-21, Department of Computer Science, State University of New York at Buffalo, Buffalo, NY, 1993.
[18] L. HEMASPAANDRA, A. NAIK, M. OGIHARA, AND A. SELMAN, *Computing solutions uniquely collapses the polynomial hierarchy*, in Proc. 5th International Symposium on Algorithms and Computation, Lecture Notes in Computer Science 834, Springer-Verlag, 1994, pp. 56–64.
[19] U. HERTRAMPF, *Relations among mod-classes*, Theoret. Comput. Sci., 74 (1990), pp. 325–328.
[20] C. JOCKUSCH, *Semirecursive sets and positive reducibility*, Trans. Amer. Math. Soc., 131 (1968), pp. 420–436.
[21] R. KARP AND R. LIPTON, *Some connections between nonuniform and uniform complexity classes*, in Proc. 12th Symposium on Theory of Computing, ACM Press, 1980, pp. 302–309. Final version: *Turing machines that take advice*, L'enseign. Math., 28 (1982), pp. 191–209.
[22] K. KO, *On self-reducibility and weak P-selectivity*, J. Comput. System Sci., 26 (1983), pp. 209–221.
[23] M. KRENTEL, *The complexity of optimization problems*, J. Comput. System Sci., 36 (1988), pp. 490–509.
[24] R. LADNER, N. LYNCH, AND A. SELMAN, *A comparison of polynomial time reducibilities*, Theoret. Comput. Sci., 1 (1975), pp. 103–123.
[25] L. LONGPRÉ AND A. SELMAN, *Hard promise problems and nonuniform complexity*, Theoret. Comput. Sci., 115 (1993), pp. 277–290.
[26] A. MEYER AND L. STOCKMEYER, *The equivalence problem for regular expressions with squaring requires exponential space*, in Proc. 13th Symposium on Switching and Automata Theory, IEEE Computer Society Press, 1972, pp. 125–129.
[27] A. NAIK, M. OGIWARA, AND A. SELMAN, *P-selective sets, and reducing search to decision vs. self-reducibility*, in Proc. 8th Conference on Structure in Complexity Theory, IEEE Computer Society Press, 1993, pp. 52–64.
[28] M. OGIWARA AND A. LOZANO, *Sparse hard sets for counting classes*, Theoret. Comput. Sci., 112 (1993), pp. 255–276.
[29] U. SCHÖNING, *Complexity and Structure*, Lecture Notes in Comp.] Sci., 211, Springer-Verlag, 1986.
[30] A. SELMAN, *P-selective sets, tally languages, and the behavior of polynomial time reducibilities on NP*, Math. Systems Theory, 13 (1979), pp. 55–65.
[31] ———, *Analogues of semirecursive sets and effective reducibilities to the study of NP complexity*, Inform. and Control, 52 (1982), pp. 36–51.
[32] ———, *Reductions on NP and P-selective sets*, Theoret. Comput. Sci., 19 (1982), pp. 287–304.
[33] J. SIMON, *On some central problems in computational complexity*, Ph.D. thesis, Cornell Univeristy, Ithaca, NY, 1975; Technical report TR75-224, Department of Computer Science, Cornell University, Ithaca, NY, 1975.
[34] L. STOCKMEYER, *The polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 1–22.
[35] T. THIERAUF, S. TODA, AND O. WATANABE, *On sets bounded truth-table reducibile to p-selective sets*, in Proc. 11th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comp. Sci. 775, Springer-Verlag, 1994, pp. 427–438.
[36] S. TODA, *On polynomial-time truth-table reducibilities of intractable sets to P-selective sets*, Math. Systems Theory, 24 (1991), pp. 69–82.
[37] L. VALIANT, *The relative complexity of checking and evaluating*, Inform. Process. Lett., 5 (1976), pp. 20–23.
[38] K. WAGNER, *The complexity of combinatorial problems with succinct input representation*, Acta Inform., 23 (1986), pp. 325–356.

SIAM J. COMPUT.
Vol. 24, No. 5, pp. 1082–1090, October 1995

009

WITH QUASILINEAR QUERIES EXP IS NOT POLYNOMIAL TIME TURING REDUCIBLE TO SPARSE SETS*

BIN FU†

Abstract. We investigate the lower bounds of queries required by the polynomial time Turing reductions from exponential time classes to the sets of small density. For complexity classes E= DTIME($2^{O(n)}$) and EXP=DTIME($2^{n^{O(1)}}$), the following results are shown in this paper: (1) For any $a < 1$, every EXP-$\leq^{\mathrm{P}}_{n^a-\mathrm{T}}$-hard set is exponentially dense. This yields EXP$\not\subseteq \mathrm{P}_{n^a-\mathrm{T}}$(SPARSE) for all $a < 1$. (2) For any $a < \frac{1}{2}$, every E-$\leq^{\mathrm{P}}_{n^a-\mathrm{T}}$-hard set is exponentially dense. (3) E$\not\subseteq \mathrm{P}_{o(n/\log n)-\mathrm{T}}$(TALLY). Our results substantially improve Watanabe's earlier theorem, E$\not\subseteq \mathrm{P}_{\log n-\mathrm{tt}}$(SPARSE) [*Proc. 2nd IEEE Conference on Structure in Complexity Theory*, 1987, pp. 138–146], [*Proc. 7th IEEE Conference on Structure in Complexity Theory*, 1992, pp. 222–238].

Key words. exponential time complexity classes, polynomial time Turing reducibilities, sparse sets

AMS subject classification. 68Q15

1. Introduction.

The study of the density of hard sets for complexity classes has a long history. Berman and Hartmanis [BH77] conjectured that all NP-$\leq^{\mathrm{P}}_m$-complete sets are isomorphic. Since all of the known NP-$\leq^{\mathrm{P}}_m$-complete sets are exponentially dense, Berman and Hartmanis proposed a weaker conjecture in which all NP-$\leq^{\mathrm{P}}_m$-complete sets are not sparse. A considerable number of previous works culminated in Mahaney [M82], which proved that no NP-$\leq^{\mathrm{P}}_m$-hard set is sparse unless P=NP. Great effort has gone into extending Mahaney's result to weaker reductions. Ogiwara and Watanabe [OW91] proved that no NP-$\leq^{\mathrm{P}}_{\mathrm{btt}}$-hard set is sparse unless P=NP.

The fact that all NP-$\leq^{\mathrm{P}}_m$-complete sets are not sparse implies P$\neq$NP. The study of the density of hard sets for NP always needs some assumption (such as P$\neq$NP). On the other hand, absolute results can be obtained for the density of hard sets for exponential time classes. In an early paper [B76], Berman studied the structure of exponential time complete sets. In [BH77], Berman and Hartmanis showed that no E-$\leq^{\mathrm{P}}_m$-hard set can be sparse. Watanabe [Wa87] strengthed this result to $\leq^{\mathrm{P}}_{\mathrm{btt}}$-hardness and his techniques can be used to show that E$\not\subseteq \mathrm{P}_{\log n-\mathrm{tt}}$(SPARSE). Separating a complexity class from a polynomial size circuit is one of the fundamental problems in complexity theory. Many people have made great efforts in this direction. It is unknown whether NEXP is computable in a polynomial size circuit. The class of languages computable by a polynomial size circuit is the same as the class of languages that is polynomial time Turing reducible to sparse sets (the class P_{T}(SPARSE)). It is generally conjectured that EXP$\not\subseteq \mathrm{P}_{\mathrm{T}}$(SPARSE) (equivalently, that EXP does not have polynomial size circuits). Karp and Lipton [KL80] showed that if EXP$\subseteq \mathrm{P}_{\mathrm{T}}$(SPARSE) then EXP=$\Sigma^p_2$. Wilson [Wi85] constructed an oracle A such that NEXP$\subseteq \mathrm{P}_{\mathrm{T}}$(SPARSE) holds relative to A. Heller [H86] constructed an oracle to collapse NEXP to BPP, which is a subclass of P_{T}(SPARSE). So, it is impossible to separate EXP from P_{T}(SPARSE) by relativizable techniques. [HOW92] and [H90] are good surveys on the recent study of sparse sets and exponential time classes.

In this paper, we show that for any $a < 1$, every EXP-$\leq^{\mathrm{P}}_{n^a-\mathrm{T}}$-hard set is exponentially dense. This yields EXP$\not\subseteq \mathrm{P}_{n^a-\mathrm{T}}$(SPARSE) for all $a < 1$. Although $\leq^{\mathrm{P}}_m$-hardness for EXP is equivalent to that for E [GH89], it does not seem to be true for $\leq^{\mathrm{P}}_{n^a-\mathrm{T}}$-hardness with $a < 1$. We show that for any $a < \frac{1}{2}$, every E-$\leq^{\mathrm{P}}_{n^a-\mathrm{T}}$-hard set is exponentially dense. Sparse languages and

*Received by the editors September 14, 1992; accepted for publication (in revised form) May 17, 1994. This research was supported in part by the 863 High Technology Plan of China (HTP863).

†Department of Computer Science, Beijing Computer Institute, Beijing 100044, People's Republic of China and Beijing Laboratory of Cognitive Science, University of Science and Technology of China. Present address: Department of Computer Science, Princeton University, Princeton, NJ 08544.

tally languages are closely related. It is well known that $P_T(SPARSE) = P_{tt}(TALLY)$ [BK88]. We also constructed a sparse set A in E such that A is $P_{o(n/\log n)-T}(TALLY)$-immune. To obtain the above results, we develop a new technique for dealing with $\leq_T^P$-reductions instead of counting the number of truth tables, which was used in the past.

Recently, Fu received a letter and a manuscript from Lutz, in which he and Mayordomo [LM] proved, using a quite different technique, that for all $a < 1$, every E-$\leq_{n^a-tt}^P$-hard set is exponentially dense.

2. Preliminaries. We use $\Sigma = \{0, 1\}$ as our alphabet. By "string" we mean an element of Σ^*; $|x|$ denotes the length of x. We use lexicographic order on Σ^*. For any strings x and y, x is smaller than y (i.e., $x < y$) if either $|x| < |y|$ or $|x| = |y|$ and there exists some k, $1 \leq k \leq |x|$, such that $(\forall i : 1 \leq i < k[x_i = y_i]$ and $[x_k = 0$ and $y_k = 1])$, where x_i is the ith symbol of the string x. For two strings $x \leq y$ in Σ^*, the interval $[x, y]$ is defined as the set $\{z : (z \in \Sigma^*)$ and $(x \leq z \leq y)\}$. For $S \subseteq \Sigma^*$, the cardinality of S is denoted by $\|S\|$. Let $S^{=n}(S^{\leq n})$ consist of all words of length $= n(\leq n)$ in S. In particular, let $\Sigma^n = \{x : x \in \Sigma^*$ and $|x| = n\}$ and $\Sigma^{\leq n} = \{x : x \in \Sigma^*$ and $|x| \leq n\}$. For a language A, χ_A is the characteristic function of A. $N = \{0, 1, 2, \ldots\}$. For a real number x, $[x]$ is the biggest integer $\leq x$. Our computation model is the Turing machine. We use the following three exponential time complexity classes:

$$E = \bigcup_{k=1}^{\infty} DTIME(2^{kn+k}),$$
$$EXP = \bigcup_{k=1}^{\infty} DTIME(2^{n^k+k}),$$
$$NEXP = \bigcup_{k=1}^{\infty} NTIME(2^{n^k+k}).$$

A $\leq_T^P$-*reduction* of A to B is a polynomial time oracle Turing machine M such that for each $x \in \sum^*$, $x \in A \Longleftrightarrow M^B$ accepts x.

For a function $g : N \longrightarrow N$, a $\leq_{g(n)-T}^P$-reduction of A to B is a polynomial time oracle Turing machine M such that $A \leq_T^P B$ is witnessed by M, and M will not query the oracle more than $g(n)$ times for each input with length n.

Let $H \subseteq \Sigma^*$, C be a class of languages and $\leq_r^P$ be a type of reduction. $P_r(H)$ is the class of all languages that are $\leq_r^P$-*reducible* to H. $P_r(C)$ is the class of languages that are $\leq_r^P$-reducible to some languages in C. If $C \subseteq P_r(H)$, then we say H is C-$\leq_r^P$-*hard*.

For language $A \subseteq \Sigma^*$, we say A is *exponentially dense* if, for some $c > 0$, $\|A^{\leq n}\| > 2^{n^c}$ for all large n. If there exists a polynomial $p(n)$ such that $\|A^{\leq n}\| < p(n)$ for all n, then we say A is *sparse*. SPARSE represents the class of all sparse languages. TALLY represents the class of all languages T with $T \subseteq \{0\}^*$.

Hartmanis [H83] introduced a "generalized Kolmogorov complexity measure." We employ this tool in the proof of our theorems.

We consider standard deterministic time-bounded Turing machines that act as transducers. We assume a standard enumeration of such transducers, say $N_1, N_2, \ldots$. For each i, let f_i be the function computed by N_i and let T_i be the running time of the transducer N_i. We assume that the enumeration has the property that there exists a universal Turing transducer N_u and a description function d with the following property: for every $i > 0$ there is a constant c_i such that for all $x \in \Sigma^*$, (a) $d(i)$ is not a prefix of $d(j)$ if $i \neq j$, (b) $f_u(d(i)x) = f_i(x)$, and (c) $T_u(d(i)x) \leq c_i \cdot T_i(x) \cdot \log T_i(x) + c_i$. We then define the following classes of strings:

$$K_i[g(n), t(n)] = \{y : \exists x[|x| \leq g(n), f_i(x) = y, \text{ and } T_i(x) \leq t(n)]\},$$

where $n = |y|$, and

$$K[g(n), t(n)] = K_u[g(n), t(n)],$$

where u denotes the index of the universal Turing machine.

We have the following fact.

LEMMA [H83]. *Let i be any index and let $g(n)$ and $t(n)$ be time-constructible functions. Then there exists $c > 0$ such that $K_i[g(n), t(n)] \subseteq K_u[g(n) + c, c \cdot t(n) \log t(n) + c]$.*

3. Number of queries and density. Suppose EXP (or E)$\subseteq \mathrm{P}_{f(n)\text{-T}}(S)$. We investigate the lower bounds for the number of queries $f(n)$ and the density of the set S in this section. Resource-bounded Kolmogorov complexity theory plays a crucial role in the proofs of our results.

For strings $w, \tau \in \Sigma^*$ and integer m, we say w is an *m-block* of τ if $\tau = w_1 \dots w_k$ for some k, $|w_1| = \cdots = |w_k| = m$, and $w = w_i$ for some integer $i : 1 \le i \le k$. The least number i with $w = w_i$ is called the *location* of w in τ. Block (m, τ) is defined as the set $\{w : w \text{ is a } m\text{-block of } \tau\}$. For $B \subseteq \Sigma^*$, Block $(m, B) = \cup_{\tau \in B}$ Block(m, τ).

The function cod : $\Sigma^* \longrightarrow \Sigma^*$ is defined by $\mathrm{cod}(x) = 1a_1 1a_2 \dots 1a_r 0^2$, where $x = a_1 a_2 \dots a_r \in \Sigma^*$. The function bin: $N \longrightarrow \Sigma^*$ is defined so that for each $i \in N$, bin(i) is the binary expression of i (for example, bin(5) = 101 and bin(8) = 1000).

PROPOSITION 1. *There exists a polynomial $p(n)$ such that for all large n, if there exist strings $\tau \in \Sigma^n$ and u, v, w, x that satisfy $\tau = uvwvx$ and $|v| > 7 \log n$, then $\tau \in K[n - 1, p(n)]$.*

Proof. Suppose n is large, $\tau \in \Sigma^n$, and $\tau = uvwvx$ with $|v| > 7 \log n$.

Let $\tau_1 = \mathrm{cod}(\mathrm{bin}(|u|))\ \mathrm{cod}(\mathrm{bin}(|v|))\ \mathrm{cod}(\mathrm{bin}(|\ w|))uvwx$. Clearly, if we have string τ_1, we can obtain u, v, w, x easily by decoding it. Thus τ can be generated by τ_1 in polynomial steps. Also, $|\tau_1| = 2|\mathrm{bin}\ (|u|)| + 2 + 2|\mathrm{bin}(|v|)| + 2 + 2|\mathrm{bin}\ (|w|)| + 2 + n - |v| \le 6 \log n + 6 + n - |v| < n - \frac{\log n}{2}$ if n is large enough. It is easy to see that there exists a polynomial $p(n)$ (which does not depend on τ) such that $\tau \in K[n - 1, p(n)]$. □

The proof of the following proposition needs the limitation $\lim_{n\to\infty}(n^n/n!)^{1/n} = e \approx 2.178$. In fact, $\ln(n!/n^n)^{1/n} = \frac{1}{n}\Sigma_{i=1}^{n} \ln(\frac{i}{n})$, whose limitation is $\int_0^1 \ln x dx = -1$. So, $n! > (\frac{n}{3})^n$ for all large n.

PROPOSITION 2. *For every $0 \le a < b$ and $\delta > 0$, $n^{(b-a+\delta)n^a} > \binom{[n^b]}{[n^a]} > n^{(b-a-\delta)n^a}$ for all large n.*

Proof. Suppose n is sufficiently large such that $2[n^a] + 2 < n^b$; $4 < n^\delta$; $[n^a]! > ([n^a]/3)^{[n^a]}$ and $[n^a]/3 > n^a/4$. Then

$$\binom{[n^b]}{[n^a]} = \frac{[n^b]([n^b]-1)\dots([n^b]-[n^a]+1)}{[n^a]([n^a]-1)\dots 1} > \frac{\left(\frac{n^b}{2}\right)^{[n^a]}}{(n^a)^{[n^a]}} = \left(\frac{n^b}{2n^a}\right)^{[n^a]} > n^{(b-a-\delta)n^a}.$$

On the other hand, $[n^b]([n^b]-1)\dots([n^b]-[n^a]+1) < (n^b)^{[n^a]}$. So

$$\binom{[n^b]}{[n^a]} < \frac{(n^b)^{[n^a]}}{\left(\frac{[n^a]}{3}\right)^{[n^a]}} < \frac{(n^b)^{[n^a]}}{\left(\frac{n^a}{4}\right)^{[n^a]}} = (4n^{b-a})^{[n^a]} \le (4n^{b-a})^{n^a} < (n^{b-a+\delta})^{n^a}. \qquad \square$$

THEOREM 3.1. *For $0 \le a < 1$ and $b > 0$, there exists a set A in* DTIME$(2^{n^{2+b}}) \cap P_{\mathrm{ctt}}$(SPARSE) *such that if A is $\le^{\mathrm{P}}_{n^a\text{-T}}$-reducible to S, then S is exponentially dense.*

Proof. Choose an integer k such that $a < 1 - \frac{1}{k}$ and $b > \frac{9}{k+1}$. We will construct a set A in DTIME$(2^{n^{2+9/(k+1)}}) \cap$ P$_{\mathrm{ctt}}$(SPARSE) such that $A \in \mathrm{P}_{n^{1-1/k}\text{-T}}(S)$ implies that S is exponentially dense.

We give an informal overview of the proof. Some ideas in [GW91] are involved in the construction of set A. Set A will be built to be exponentially dense. Each string in A is hard to compress and hard to describe relative to other strings in A. Suppose S is of small density and A is Turing reducible to S via oracle Turing machine M bounded by a small number of queries in at most $p(n)$ time. For all inputs in Σ^n, every string queried by M^S is of length $\leq p(n)$. Partition $\Sigma^{\leq p(n)}$ into some intervals such that for any $x \leq y$, x, y are in the same interval iff for any z: $x \leq z \leq y$, $\chi_S(x) = \chi_S(y) = \chi_S(z)$. Thus, the number of intervals is not more than $2\,\|S^{\leq p(n)}\| + 1$. Since both the density of S and the number of queries of M^S are small, there exists a subset B of $A^{=n}$ such that (i) B contains many elements, (ii) for all inputs in B, M^S will query the same number t of questions, and (iii) all of the ith queries of M^S with inputs in B are in the same interval of $\Sigma^{\leq p(n)}$ for $i \leq t$.

Let α_i be the answer of oracle S for the ith query of M^S with an input in B. Strings $x_1, \ldots, x_t, y_1, \ldots, y_t$ are chosen from B such that for each input string z in B, the ith string queried by M^S with input z is located between the two ith strings queried by M^S for inputs x_i, y_i, respectively. If we have $\alpha_1, \ldots, \alpha_t, x_1, \ldots, x_t, y_1, \ldots, y_t$, then we can generate all of the strings in B by simulating M (with no oracle) on all of the strings in Σ^n. Since the number of queries t is small and the density of B is large, we have the intuition that the information content of $\alpha_1, \ldots, \alpha_t, x_1, \ldots, x_t, y_1, \ldots, y_t$ is small and that of B is high. This is a contradiction.

First, we construct the set A as follows.

CONSTRUCTION OF A_n.

Case 1. $n \neq m^{k+1}$ for all integers m.

Set $A_n = \emptyset$.

Case 2. $n = m^{k+1}$ for some integer m.

Choose the least string $\tau \in \Sigma^{n^{2+8/(k+1)}} (= \Sigma^{mn^{2+7/(k+1)}})$ such that $\tau \notin K[n^{2+8/(k+1)} - 1, 2^{100n}]$ (by Proposition 1, if m is large, then all m-blocks of τ are different from each other).

Set $A_n = \{u_1 u_2 \ldots u_{m^k} : \text{each } u_j \text{ is a } m\text{-block of } \tau \text{ and } u_1 < u_2 < \cdots < u_{m^k}\}$.

Let $A = \cup_{n=1}^{\infty} A_n$.

CLAIM 1. $A \in \mathrm{P}_{\mathrm{ctt}}(\mathrm{SPARSE})$.

Proof. Let τ_n be the string as in τ in the construction of A_n (if $n \neq m^{k+1}$, then τ is the empty string). Let $S_0 = \cup_{m=0}^{\infty} \mathrm{Block}(m, \tau_{m^{k+1}})$. It is easy to verify that S_0 is sparse and $A \leq^{\mathrm{P}}_{\mathrm{ctt}} S_0$. □

CLAIM 2. $A \in \mathrm{DTIME}(2^{n^{2+9/(k+1)}})$.

Proof. For each $x \in \Sigma^*$, $x \in A$ iff $x \in A_n$, where $n = |x|$.

Case 1. If $n \neq m^{k+1}$ for all m, then $A_n = \emptyset$. Thus, $x \notin A$.

Case 2. If $n = m^{k+1}$ for some integer m, the time consumption of Case 2 of stage n is bounded by $2^{101n} \cdot 2^{n^{2+8/(k+1)}} < 2^{n^{2+9/(k+1)}}$ if n is large enough. Clearly the question of whether $x \in A$ can be determined in $2^{n^{2+9/(k+1)}}$ steps for all large n. □

Suppose $A \leq^{\mathrm{P}}_{n^{1-1/k}\text{-T}} S$ and S is not exponentially dense. Let $A \leq^{\mathrm{P}}_{n^{1-1/k}\text{-T}} S$ via polynomial time oracle Turing machine M. Without loss of generality, we assume that M asks exactly $[n^{1-1/k}]$ questions. Let $p(n)$ be a polynomial with positive coefficients such that $p(n) > n$ and M will stop in at most $p(n)$ steps for each input of length n.

Choose integer d such that $p((n+1)^{k+1}) < n^d$ for all $n \geq 2$. Since S is not exponentially dense, there exist infinitely many n's such that $\|S^{\leq n}\| < 2^{n^{1/dk(k+1)}}$.

Fix integers n_0, n, and m large enough to satisfy the following conditions:

(a) $\|S^{\leq n_0}\| < 2^{n_0^{\frac{1}{dk(k+1)}}}$,

(b) $n \geq 2$,

(c) $n = m^{k+1}$,

(d) $p(n) \leq n_0$,

(e) $n_0 < p((m+1)^{k+1})$,

(f) $n^{(1+\frac{8}{k+1}-\frac{1}{k+1})n^{1-\frac{1}{k+1}}} < \binom{n^{2+\frac{7}{k+1}}}{n^{1-\frac{1}{k+1}}}$ and $\binom{n^2}{n^{1-\frac{1}{k+1}}} < n^{(1+\frac{1}{k+1}+\frac{1}{k+1})n^{1-\frac{1}{k+1}}}$

(by Proposition 2, where $\delta = \frac{1}{(k+1)}$),

(g) $30 + 30\log n < n^{\frac{1}{k+1}}$,

(h) $2 \cdot 2^{n^{\frac{1}{k(k+1)}}} + 1 < n^{n^{\frac{1}{k(k+1)}}}$.

Note that

$$\|S^{\leq p(n)}\| \leq \|S^{\leq n_0}\| \leq 2^{n_0^{\frac{1}{dk(k+1)}}} < 2^{n^{\frac{1}{k(k+1)}}}$$

(because $p(n) \leq n_0 < p((m+1)^{k+1}) < p((n+1)^{k+1}) < n^d$).

For each input $x \in \Sigma^n$, the strings queried by M^S are of length $\leq p(n)$. We partition $\Sigma^{\leq p(n)}$ into *intervals* $U_1, U_2, \ldots$ such that for $x \leq y$ in $\Sigma^{\leq p(n)}$, x and y are in the same interval if and only if for every z in $[x, y]$, $\chi_S(x) = \chi_S(y) = \chi_S(z)$.

Therefore, $\Sigma^{\leq p(n)}$ can be partitioned into at most $2\,\|S^{\leq p(n)}\| + 1$ intervals. Since M asks the oracle for all inputs of length n exactly $[n^{1-1/k}]$ times, let $t = [n^{1-1/k}]$ be the number of queries of M for input of length n.

For input $x \in \Sigma^n$, we define $\text{Query}_{M^S}(x) =< M(x, 1), \ldots, M(x, t) >$, where $M(x, i)$ is the ith string queried by M^S.

For $x, y \in \Sigma^n$, we say $\text{Query}_{M^S}(x)$ is *similar* to $\text{Query}_{M^S}(y)$ if and only if $M(x, i)$ and $M(y, i)$ are in the same interval U_j for all $i = 1, \ldots, t$.

Partition Σ^n into blocks $B_1, B_2, \ldots$ such that for any $x, y \in \Sigma^n$, x, y are in the same block iff $\text{Query}_{M^S}(x)$ is similar to $\text{Query}_{M^S}(y)$. Since for each input in Σ^n, M^S queries the oracle $[n^{1-1/k}]$ times, Σ^n contains at most

$$(2\,\|S^{\leq p(n)}\| + 1)^{n^{1-\frac{1}{k}}} \leq (2 \cdot 2^{n^{\frac{1}{k(k+1)}}} + 1)^{n^{1-\frac{1}{k}}} < (n^{n^{\frac{1}{k(k+1)}}})^{n^{1-\frac{1}{k}}} = n^{n^{\frac{1}{k(k+1)}+1-\frac{1}{k}}} = n^{n^{1-\frac{1}{k+1}}}$$

blocks.

By Case 2 of stage n in the construction, τ has $n^{2+7/(k+1)}$ m-blocks. By (f), we have

$$\|A_n\| = \binom{n^{2+\frac{7}{k+1}}}{m^k} = \binom{n^{2+\frac{7}{k+1}}}{n^{1-\frac{1}{k+1}}} > n^{(1+\frac{8}{k+1}-\frac{1}{k+1})n^{1-\frac{1}{k+1}}} = n^{(1+\frac{7}{k+1})n^{1-\frac{1}{k+1}}}.$$

Thus, there exists a block B_i such that B_i contains more than

$$\frac{n^{(1+\frac{7}{k+1})n^{1-\frac{1}{k+1}}}}{n^{n^{1-\frac{1}{k+1}}}} = n^{(1+\frac{6}{k+1})n^{1-\frac{1}{k+1}}}$$

elements in A_n. Let $B = B_i \cap A_n$. Thus,

$$\|B\| > n^{(1+\frac{6}{k+1})n^{1-\frac{1}{k+1}}}.$$

Since B has more than $n^{(1+6/(k+1))n^{1-1/(k+1)}}$ strings, it follows that Block(m, B) contains more than n^2 m-blocks of the string τ chosen in the construction of A_n (because

$$\binom{n^2}{m^k} = \binom{n^2}{n^{1-\frac{1}{k+1}}} < n^{(1+\frac{1}{k+1}+\frac{1}{k+1})n^{1-\frac{1}{k+1}}} = n^{(1+\frac{2}{k+1})n^{1-\frac{1}{k+1}}}$$

by (f)).

Now choose the strings $x_1, \ldots, x_t, y_1, \ldots, y_t$ from B such that for any $u \in B$, $M(x_i, i) \leq M(u, i) \leq M(y_i, i)$, $i = 1, \ldots, t$. Let $\alpha_i = \chi_S(M(x_i, i))$, $i = 1, \ldots, t$. We will use $\alpha_1, \ldots, \alpha_t, x_1, \ldots, x_t, y_1, \ldots, y_t$ to generate all of the strings in B.

For each $u \in \Sigma^n$, the question of whether u is in B can be determined by the following algorithm.

ALGORITHM (The reader should note that the algorithm will use the information of $\alpha_1, \ldots, \alpha_t, x_1, \ldots, x_t, y_1, \ldots, y_t$, but it does not ask the oracle S directly.)

Input u with length n.

$i := 1$

Repeat

Simulate the computation of M (between the $(i-1)$th query and the ith query) with input u until M makes the ith query ($M(u, i) \in S$?).

Simulate the computation of M with inputs x_i, y_i, respectively, to get the strings $M(x_i, i)$ and $M(y_i, i)$ (Since we have $\alpha_1, \ldots, \alpha_t$, we know $\chi_S(M(x_i, j)) = \chi_S(M(y_i, j)) = \alpha_j$ for all $j < i$. Let M get answer α_j at its jth query for all $j < i$. Thus, we can obtain $M(x_i, i)$ and $M(y_i, i)$ in polynomial steps.)

If $M(x_i, i) \leq M(u, i) \leq M(y_i, i)$

then $i := i + 1$ and let M get the answer α_i from the oracle

else reject u and exit.

until $i = t + 1$.

Accept u iff M accepts u.

CLAIM 3. $u \in B \Longleftrightarrow u$ will be accepted by the algorithm.

Proof. $\Longrightarrow$ Suppose $u \in B$; note that $\text{Query}_{M^S}(u)$ is similar to $\text{Query}_{M^S}(v)$ for all v in B. From the definition of $\alpha_1, \ldots, \alpha_t, x_1, \ldots, x_t, y_1, \ldots, y_t$, $M(x_i, i) \leq M(u, i) \leq M(y_i, i)$ for all $i \leq t$. Since M^S accepts u (because $u \in B \subseteq A$), u will be accepted by the algorithm.

$\Longleftarrow$ Suppose u is accepted by the algorithm. It is easy to see that for all $i \leq t$, $M(x_i, i) \leq M(u, i) \leq M(y_i, i)$. So, $\text{Query}_{M^S}(u)$ is similar to $\text{Query}_{M^S}(x)$ for all x in B. Since u is accepted by the algorithm, u will be accepted by M^S. Thus $u \in B$. □

Therefore, if we have $\alpha_1, \ldots, \alpha_t, x_1, \ldots, x_t, y_1, \ldots, y_t$, then we can get all elements in Block(m, B) within 2^{2n} steps.

We will show that τ can be generated by a string of length $< n^{2+8/(k+1)} - n^2$ in 2^{4n} steps.

Let string $\xi = \text{cod}(\text{bin}(n))\alpha_1 \ldots \alpha_t x_1 \ldots x_t y_1 \ldots y_t \text{cod}(\text{bin}(e_1)) \ldots \text{cod}(\text{bin}(e_z))\tau'$, where $e_1, \ldots, e_z$ and τ' are defined as follows.

Let $w^{(1)} < \cdots < w^{(z)}$ be all of the elements in Block(m, B). Since $z = \|\text{Block}(m, B)\|$, we must have $n^2 \leq z \leq n^{2+7/(k+1)}$. For each m-block $w^{(i)}$ of τ in Block(m, B), let e_i be the location of $w^{(i)}$ in τ (this means $w^{(i)} = w_{e_i}$, where $\tau = w_1 \ldots w_{n^{2+7/(k+1)}}$ and each w_i is an m-block of τ).

Let τ' be the concatenation (preserving the order in τ) of the m-blocks of τ in Block(m, τ) − Block(m, B). That is, $\tau' = w_{j_1} \ldots w_{j_{n^{2+7/(k+1)}-z}}$, and $k_1 < k_2$ implies that $j_{k_1} < j_{k_2}$, where $\tau = w_1 \ldots w_{n^{2+7/(k+1)}}$.

CLAIM 4.

$$|\xi| < n^{2+\frac{8}{k+1}} - n^2.$$

Proof. $|\xi| = 2\log n + 2 + t + 2t \cdot n + \sum_{i=1}^{z} |\mathrm{cod}(\mathrm{bin}(e_i))| + |\tau'| < 7tn + z(2\log n + 2) + (n^{2+7/(k+1)} - z)m < 7z + 12z\log n + (n^{2+7/(k+1)} - z)m = n^{2+7/(k+1)}m - (m - 12\log n - 7)z < n^{2+8/(k+1)} - n^2$ (by inequality (g)). □

CLAIM 5. τ can be generated by ξ in 2^{4n} steps.

Proof. If we have the string

$$\xi = \mathrm{cod}(\mathrm{bin}(n))\alpha_1 \ldots \alpha_t x_1 \ldots x_t y_1 \ldots y_t \mathrm{cod}(\mathrm{bin}(e_1)) \ldots \mathrm{cod}(\mathrm{bin}(e_z))\tau,$$

we can obtain $\mathrm{cod}(\mathrm{bin}(n))$. Furthermore, we get n and $t = [n^{1-1/k}]$. We can obtain $\alpha_1, \ldots, \alpha_t, x_1, \ldots, x_t, y_1, \ldots, y_t, e_1, \ldots e_z$ and τ'. Thus, within 2^{3n} steps we can generate all m-blocks of τ in $\mathrm{Block}(m, B)$. From the $e_1, \ldots, e_z$ we can know the location of each element of $\mathrm{Block}(m, B)$ in τ. Thus τ can be generated by ξ. The above computation can be finished in 2^{4n} steps. □

Therefore, τ is generated by a string of length $< n^{2+8/(k+1)} - n^2$ in 2^{4n} steps. Thus, $\tau \in K[n^{2+8/(k+1)} - 1, 2^{100n}]$. This is a contradiction.

COROLLARY 3.2. *For any* $0 \le a < 1$, *every* EXP-$\le^{\mathrm{P}}_{n^a-\mathrm{T}}$*-hard set is exponentially dense.*

COROLLARY 3.3. *For any* $0 \le a < \frac{1}{2}$, *every* E-$\le^{\mathrm{P}}_{n^a-\mathrm{T}}$*-hard set is exponentially dense.*

Proof. From Theorem 3.1, we know that for each integer k, there exists a language $A \in \mathrm{DTIME}(2^{n^{2+1/k}})$ such that $A \in \mathrm{P}_{n^{1-1/k}-\mathrm{T}}(S)$ implies that S is exponentially dense.

Suppose $\mathrm{E} \subseteq \mathrm{P}_{n^a-\mathrm{T}}(S)$ for some $a < \frac{1}{2}$. Let k be an integer such that $a < \frac{1}{2} - \epsilon$, where $\epsilon = \frac{1}{4k}$.

Let $A \in \mathrm{DTIME}\,(2^{n^{2+1/k}})$ such that $A \in \mathrm{P}_{n^{1-1/k}-\mathrm{T}}(S)$ implies that S is exponentially dense. Let $A_1 = \{\mathrm{pad}(x) : x \in A \text{ and } |x| = n\}$, where $\mathrm{pad}(A) = x10^{n^{2+1/k}-n-1}$. Clearly, $A_1 \in \mathrm{E}$. So $A_1 \in \mathrm{P}_{n^{1/2-\epsilon}-\mathrm{T}}(S)$. Therefore,

$$A \in \mathrm{P}_{n^{(\frac{1}{2}-\epsilon)(2+\frac{1}{k})}-\mathrm{T}}(S) = \mathrm{P}_{n^{1-\frac{1}{4k^2}}-\mathrm{T}}(S) \subseteq \mathrm{P}_{n^{1-\frac{1}{k}}-\mathrm{T}}(S).$$

Hence, S is exponentially dense. □

COROLLARY 3.4. $P_{\mathrm{ctt}}(\mathrm{SPARSE}) \not\subseteq P_{n^a-\mathrm{T}}(\mathrm{SPARSE})$ *for all* $a < 1$.

The obstacle for improving Corollary 3.2 to the case $a = 1$ is the fact that the number of blocks $B_1, B_2, \ldots$ will be greater than the number of elements in A_n if M is allowed to ask the oracle n times. So this cannot guarantee that there is a block B_i to contain exponential elements in A_n. Improving Corollary 3.3 to $a < 1$ may be possible if we can find another way with information content less than $\alpha_1, \ldots, \alpha_t, x_1, \ldots, x_t, y_1, \ldots, y_t$ to generate all of the strings in B. Unfortunately, so far we have no more efficient way of replacing $\alpha_1, \ldots, \alpha_t, x_1, \ldots, x_t, y_1, \ldots, y_t$, which have information content of nearly n^2 (in fact, it is $O(n^{2-1/k})$) bits.

Let C be a class of languages and $A \subseteq \Sigma^*$. If A is infinite and contains no infinite subsets in C, then we say that A is C-immune. For functions $f, g : N \longrightarrow N$, we say $f(n)$ is $o(g(n))$ if, for every $c > 0$, $f(n) < cg(n)$ for all large n.

Book and Ko [BK88] showed that $\mathrm{P_T}(\mathrm{SPARSE}) = \mathrm{P_T}(\mathrm{TALLY})$. Theorem 3.1 says that each sparse set is not EXP-$\le^{\mathrm{P}}_{n^a-\mathrm{T}}$-hard for all $a < 1$. For tally sets, a better lower bound than $n^a (a < 1)$ will be obtained in the following theorem.

THEOREM 3.5. *There exists a sparse set* A *in* E *such that* A *is* $\mathrm{P}_{o(n/\log n)-\mathrm{T}}(\mathrm{TALLY})$*-immune.*

Proof. For each integer n, let a_n be the least string in Σ^n such that $a_n \notin K[n-1, 2^{2n}]$.

Let $A = \{a_1, \ldots, a_n, \ldots\}$. Clearly, A is a sparse set in E. Suppose that A contains an infinite subset $A_1 \in \mathrm{P}_{o(n/\log n)-\mathrm{T}}(\mathrm{TALLY})$. Let $A_1 \le^{\mathrm{P}}_{o(n/\log n)-\mathrm{T}} T$ via the polynomial time oracle Turing machine M such that M queries the oracle $o(\frac{n}{\log n})$ times, where $T \subseteq \{0\}^*$. Let

$p(n)$ be a polynomial which is the time bound for M. It is convenient to assume that each string queried by M is in $\{0\}^*$.

We choose a large n such that $a_n \in A_1$. Let $x_1, \ldots, x_t$ be the strings queried by M^T with input a_n. Let $\alpha_i = \chi_T(x_i), i = 1, \ldots, t$. We use $\beta = \text{cod}(\alpha_1 \ldots \alpha_t)\text{cod}(\text{bin}(|x_1|))\text{cod}(\text{bin}(|x_2|)) \ldots \text{cod}(\text{bin}(|x_t|))$ to code $\alpha_1 \ldots \alpha_t, x_1, \ldots, x_t$. Clearly, $|\beta| = o(n)$. So, $|\beta| < \frac{n}{2}$ if n is large. □

The following algorithm will output a_n in 2^{2n} steps with input β.

ALGORITHM (The algorithm will use the information of β, but it does not ask the oracle T directly.)

Input β
Decode β to get $t, \alpha_1, \ldots, \alpha_t, x_1, \ldots, x_t$
For each $x \in \Sigma^n$ do
begin
$i := 1$
Repeat
Simulate the computation of M (between the $(i-1)$th query and ith query) on input x until M makes the ith query ($M(x, i) \in T$?).
If $M(x, i) = x_i$
then $i := i + 1$ and let M get the answer α_i from the oracle.
else reject x
Until $(i > t)$ or (x is rejected)
If M asks exactly t questions and accepts x **then** output x and exit.
end

Since $a_n \notin K[n-1, 2^{2n}]$, this is a contradiction. □

COROLLARY 3.6. $E \not\subseteq P_{o(n/\log n)-T}(\text{TALLY})$.

COROLLARY 3.7. $E \not\subseteq P_{n^a-T}(\text{TALLY})$ *for all* $a < 1$.

COROLLARY 3.8. $\text{SPARSE} \not\subseteq P_{o(n/\log n)-T}(\text{TALLY})$.

Corollary 3.8 is in contrast to Buhrman, Longpre, and Spaan's [BLS92] recent theorem, $\text{SPARSE} \subseteq P_{ctt}(\text{TALLY})$.

The set A constructed in the proof of Theorem 3.5 has the property that $\|A^{=n}\| = 1$. It is easy to prove that for any set, A with $\|A^{=n}\| = O(1)$ is $\leq^P_{O(n/\log n)-tt}$-reducible to tally sets. So more complicated sparse set should be constructed if we want to improve Theorem 3.5.

Our techniques seems unable to settle the following open problem: Does $\text{EXP} \not\subseteq P_{n^a-T}(\text{SPARSE})$ for $a = 1$?

Acknowledgments. The author is grateful to Steven Homer, Feng Li, Qiongzhang Li, Shouwen Tang, Dingkang Wang, and Jie Wang for their help and encouragement for this research. The author also thanks Hong-Zhou Li and Jack H. Lutz for their very helpful suggestions for improving our writing. In addition, the author thanks Ker-I Ko and two unknown referees for their valuable comments on improving the presentation.

REFERENCES

[B76] L. BERMAN, *On the structure of complete sets*, in Proc. 17th IEEE Conference on Foundations of Computer Science, Berkeley, CA, 1976, pp. 76–80.

[BH77] L. BERMAN AND J. HARTMANIS, *On isomorphisms and density of NP and other complete sets*, SIAM J. Comput., 1 (1977), pp. 305–322.

[BK88] R. BOOK and K. KO, *On sets truth-table reducible to sparse sets*, SIAM J. Comput., 17 (1988), pp. 903–919.

[BORW88] R. Book, P. Orponen, D. Russo, and O. Watanabe, *Lowness properties of sets in the exponential time hierarchy*, SIAM J. Comput., 17 (1988), pp. 504–516.

[BLS92] H. Buhrman, L. Longpre, and E. Spaan, *Sparse reduces conjuctively to tally*, in Proc. 6th IEEE Conference on Structure in Complexity Theory, San Diego, CA, 1993, pp. 208–214.

[GH89] K. Ganesan and S. Homer, *Complete problems and strong polynomial reducibilities*, Theoretical Aspects of Computer Science, Lecture Notes in Computer Science 349, Springer-Verlag, Berlin, New York, 1989, pp. 240–250.

[GW91] R. Gavalda and O. Watanabe, *On the computational complexity of small descriptions*, in Proc. 6th IEEE Conference on Structure in Complexity Theory, Chicago, IL, 1991, pp. 89–101.

[H83] J. Hartmanis, *Generalized Kolmogorov complexity and the structure of feasible computations*, in Proc. 24th IEEE Symposium on Foundations of Computer Science, Tuscon, AZ, 1983, pp. 439–445.

[HS65] J. Hartmanis and R. Stearns, *On computational complexity of algorithms*, Trans. Amer. Math. Soc., 117 (1965), pp. 285–306.

[H86] H. Heller, *On relativized exponential and probabilistic complexity classes*, Inform. and Control, 71 (1986), pp. 213–243.

[HOW92] L. A. Hemachandra, M. Ogiwara, and O. Watanabe, *How hard are sparse sets?*, in Proc. 7th IEEE Conference on Structure in Complexity Theory, Boston, MA, 1992, pp. 222–238.

[H90] S. Homer, *Structural properties of nondeterministic complete sets*, in Proc. 5th IEEE Conference on Structure in Complexity Theory, Barcelona, Spain, 1990, pp. 3–10.

[KL80] R. Karp and R. Lipton, *Some connections between nonuniform and uniform complexity classes*, in Proc. 12th ACM Symposium on Theory of Computing, 1980, pp. 302–309.

[LM] J. Lutz and E. Mayordomo, *Measure, stochasticity, and the density of hard languages*, SIAM J. Computing, 23 (1994), pp. 762–779.

[M82] S. Mahaney, *Sparse complete sets for NP: Solution of a conjecture of Berman and Hartmanis*, J. Comput. System Sci., 25 (1982), pp. 130–143.

[OW91] M. Ogiwara and O. Watanabe, *On polynomial-time bounded truth-table reducibility of NP to sparse sets*, SIAM J. Comput., 20 (1991), pp. 471–483.

[Wa87] O. Watanabe, *Polynomial time reducibility to a set of small density*, in Proc. 2nd IEEE Conference on Structure in Complexity Theory, Ithaca, NY, 1987, pp. 138–146.

[Wi85] C. B. Wilson, *Relativized circuit complexity*, J. Comput. System Sci., 31 (1985), pp. 169–181.

SIAM J. COMPUT.
Vol. 24, No. 5, pp. 1091–1103, October 1995

010

NEW TIGHT BOUNDS ON UNIQUELY REPRESENTED DICTIONARIES*

ARNE ANDERSSON† AND THOMAS OTTMANN‡

Abstract. We present a solution to the dictionary problem, where each subset of size n of an ordered universe is represented by a unique structure containing a (unique) binary search tree. The structure permits the execution of *search*, *insert*, and *delete* operations in $O(n^{1/3})$ time in the worst case. We also give a general lower bound, stating that for any unique representation of a set in a graph of bounded out-degree, one of the operations *search* or *update* must require a cost of $\Omega(n^{1/3})$. Therefore, our result sheds new light on previously claimed lower bounds for the unique representation of dictionaries.

Key words. analysis of algorithms, data structures, dictionary problem, uniquely represented dictionaries, binary search trees

AMS subject classifications. 68P05, 68P10, 68P20, 68Q05, 68Q25, 68R10

1. Introduction.

A *dictionary* is a set of items on which *search*, *insert*, or *delete* operations can be performed. The *dictionary problem* asks for a family of data structures to store the sets of items and for algorithms to carry out the dictionary operations efficiently. We consider a data structure as a graph consisting of nodes linked together by pointers; one item is stored in each node. The nodes represent the storage locations. Pointer paths correspond to access paths for the stored items.

In general, there can be many different structures which store the same set of items. We may perform a sequence of insert and delete operations starting with an initially empty structure to obtain a structure storing a given set of items. The shape of the obtained structure may depend on the size of the stored set, the set itself (but not only on its size), or the generation history.

We call a dictionary *set-uniquely* represented, if each *set* of items is represented by a unique data structure. In other words, for each set of items there is only one possible graph that represents the set. We call a dictionary *size-uniquely* represented if each set of the same *size* is represented by the same structure. Note that each size-unique representation is also set-unique. We also require that the values stored in the nodes are constrained by a fixed (for any graph) total order, that is, the representation is *order-unique*. The *unique representation problem* for dictionaries asks for efficient algorithms for maintaining a set- or size-unique representation of dictionaries.

A simple example of a size-unique representation of dictionaries is a sorted linked list. The randomized search trees of Aragon and Seidel [1] (with the random generator replaced by a universal hash function) are an example of a set-unique representation of dictionaries which is not size-unique.

It seems that deterministic solutions of the unique representation problem require more time for at least one of the three dictionary operations. Thus, a large variety of different structures for storing the same set or sets of the same size seems to be the price that must be paid for fast search and update. This observation was already made quite early by Snyder [5]. He shows that $\Omega(\sqrt{n})$ time is necessary for at least one of the three dictionary operations if

*Received by the editors December 7, 1992; accepted for publication (in revised form) May 20, 1994. A preliminary version of this paper appeared in the Proceedings of the IEEE Foundations of Computer Science, San Juan, Puerto Rico, 1991. This work was supported by grants from the National Swedish Board for Technical Development and from the Deutsche Forschungsgemeinschaft (DFG Ot 64/8–1).

†University of Lund, Department of Computer Science, Box 118, S-22100 Lund, Sweden (arne@dna.lth.se).

‡Universität Freiburg, Institut für Informatik, Rheinstr. 10–12, 79104 Freiburg, Germany (ottmann@informatik.uni-freiburg.de).

the dictionary is size-uniquely represented by a tree-like search structure. He also presents a structure which he calls the "jelly-fish," for which each of the three dictionary operations, search, insert, and delete, can be performed in time $O(\sqrt{n})$. However, the computational model underlying Snyder's lower bound proof and the operations used in the update algorithms for the jelly-fish structure are not the same. Hence, as we will show in this paper, there is room for improvement.

Recently, Sundar and Tarjan [6] studied the unique representation problem in a different context. They use a nondestructive CONS operation as the only primitive for creating and changing trees and they show that $\Theta(\sqrt{n})$ CONS operations are necessary and sufficient for maintaining unique binary search trees.

In this paper we present new size- and order-unique representations of dictionaries that allow us to perform search operations in time $O(c \cdot n^{1/c})$ and updates in time $O(\sqrt{n})$ when c is even, and $O(n^{(c-1)/2c})$ when c is odd, for any $c \geq 2$. As primitives for creating and changing structures, we allow pointer changes and the creation and disposition of nodes as Snyder [5] did. Similar to the structures of Snyder [5] and of Sundar and Tarjan [6], our structures are not "pure" tree structures but can be viewed as trees embedded in graphs. We also give a lower bound stating that $\Omega(n^{1/3})$ time is necessary per operation when a dictionary is size- and order-uniquely represented in a directed graph of bounded out-degree, and pointer changes are the basic primitives for manipulating graphs.

Hence, we show that there is a huge gap between unique and nonunique representations of dictionaries. We feel that these findings point to a fundamental fact in the theory of data structures.

2. Model of computation. We consider size- and order-unique representations of dictionaries by graphs of bounded out-degree ($\leq k$) and assume that for a given n there is only one graph of n nodes. Furthermore, we assume that for each graph the nodes are constrained by a fixed total order. The elements of a given set of size n are stored in the nodes of the graph in such a way that the ith element is stored in the ith node for each i.

Each search starts at one specified node, called the *root*, and follows a number of edges until the searched element is found or the search ends unsuccessfully, because some termination condition has become true. All elements must be reachable from the root, and hence each node (except the root) has to have at least one incoming edge. The cost of a search equals the number of traversed edges plus one.

When performing an update, a graph may be changed by one of the following operations:

- create/remove a node;
- change/add/remove one outgoing edge from a node (pointer change);
- exchange elements between two nodes.

Each operation requires a cost of $\Theta(1)$. After a creation the node contains an element and has no outgoing edges. (Since the graph has its out-degree bounded by a constant k, we may add k outgoing edges in constant time.)

It should be pointed out that this cost somewhat underestimates the real cost of pointer changes and element exchanges, since we do not include the time required to locate the node where a pointer change or an exchange has to be performed. However, when presenting our upper bound, we will not hide any costs by this simplification.

3. Semidynamic c-level jump lists. Snyder [5] introduced the jelly-fish as a size-unique representation of dictionaries. We first propose a slightly different view of his structure. It leads to the 2-level jump list, which has the same $O(\sqrt{n})$ worst-case time bound for all three dictionary operations. To simplify the presentation of 2-level jump lists we assume that $i^2 \leq n < (i+1)^2$. Hence, we assume that the size n of the dictionary does not vary unboundedly

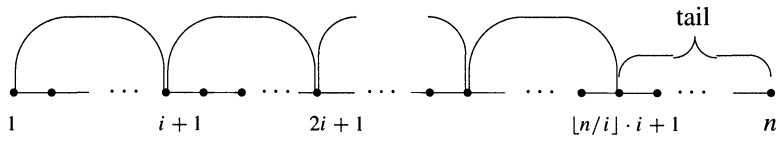


FIG. 1. *2-level jump list of size n.*

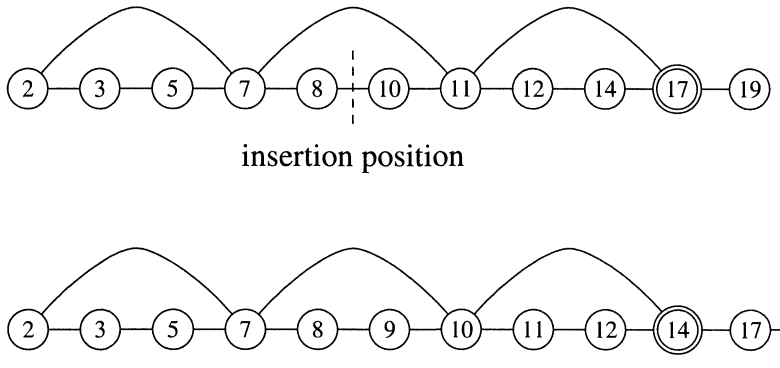


FIG. 2. *Insertion of 9 into a 2-level jump list.*

under insertions and deletions, but stays in between the given bounds $i^2 \leq n < (i+1)^2$ for some fixed i.

A *2-level jump list of size n* consists of a doubly linked list of n nodes $1, \ldots, n$; that is, for each p, $1 \leq p < n$, the nodes p and $p+1$ are linked together by a pair of *1-level links*. We call the sequence of nodes linked together by 1-level links the *1-level list*. Furthermore, the nodes $1, i+1, 2i+1, \ldots, \lfloor n/i \rfloor \cdot i + 1$ are linked together into the 2-level list, which we also call the *top-level list*. Figure 1 displays the structure of 2-level jump lists.

We require that the elements of a set of n items be stored in ascending order in the nodes $1, 2, \ldots, n$. Thus, we obtain a size- and order-unique representation of dictionaries.

It should be clear how to *search* for an item by performing at most $2i$ key comparisons: use the top-level list to determine the sequence of at most i nodes which may contain the item and perform a linear search among them following 1-level pointers. As long as n stays in the range $i^2 \leq n < (i+1)^2$, *updates* can also be performed in $O(i)$ steps: First, determine the position of the item which has to be inserted or deleted in the 1-level list. This takes $O(i)$ steps. Then insert (or delete) the element in the 1-level list. This is a constant time operation that has the effect that one sequence of nodes in the 1-level list spanned by a top-level-list pointer has become either too long by one (after an insertion) or too short by one (after a deletion). Therefore, some pointers of the top-level list have to be shifted by one position to the left or one position to the right.

Figure 2 show an example of an insertion of item 9 into a 2-level jump list of size 11 storing the set $\{2, 3, 5, 7, 8, 10, 11, 12, 14, 17, 19\}$. Note that an insertion increases the length of the tail of the 2-level jump list by one. Therefore, the top-level list has to grow by one element as soon as the length of the tail exceeds i. Similarly, a deletion may require the shortening of the top-level list by one element. Adjusting the top-level list after an insertion or deletion takes time $O(i)$ in the worst case.

So far, 2-level jump lists are only semidynamic, because we did not allow n to vary freely. It is not difficult to see that the structure can be made fully dynamic without destroying its

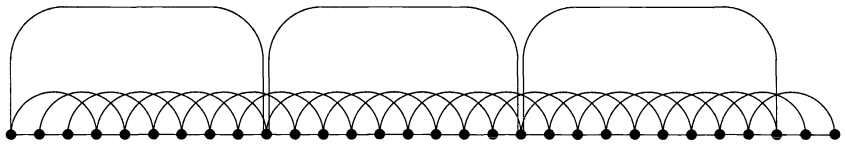


FIG. 3. *3-level jump list of size* 30.

features. Maintain a pair of structures, a primary structure and a shadow structure, corresponding to two adjacent ranges of n. Smoothly construct or destroy both structures during insertions or deletions, and switch between the structures appropriately. Whenever n exceeds the upper bound of the interval $i^2 \leq n < (i+1)^2$, we must have a 2-level jump list of size n ready with top-level pointers of length $(i+1)$ instead of length i. Similarly, whenever n falls below i^2, a 2-level jump list with top-level pointers of length $(i-1)$ must be available. Because we can spend additional time i per update operation to construct (respectively, destroy) part of the structures for n in one of the neighboring intervals $(i+1)^2 \leq n < (i+2)^2$ or $(i-1)^2 \leq n < i^2$, and because the length of the interval $i^2 \leq n < (i+1)^2$ is $\Theta(i)$, we have time $\Theta(i^2) = \Theta(n)$ to construct the desired structures. This is certainly sufficient. We avoid a more detailed description, because we will show in the next section that a much more elegant solution to the problem of fully dynamizing the structure is possible.

The 2-level jump list and Snyder's jelly-fish are similar. The jelly-fish consists of a binary search tree of size $\Theta(\sqrt{n})$ where each leaf is associated with a circular list of size $\Theta(\sqrt{n})$; cf. Fig. 9(a); the 2-level jump list is illustrated in Fig. 9(c). A minor difference is that the tree is replaced by a linked list; a major difference is that in our structure no nodes are distinguished as leaves.

We will now introduce c-level jump lists for every $c \geq 3$ as natural generalizations of 2-level jump lists. For convenience let us assume that $i^c \leq n < (i+1)^c$ for some fixed i. A *c-level jump list of size n* consists of n nodes $1, \ldots, n$. The nodes are linked together by pointers, arranged in levels:

Lower levels. For each j, $1 \leq j \leq \lceil c/2 \rceil$ and each p, $1 \leq p \leq n - i^{j-1}$, the nodes p and $p + i^{j-1}$ are linked together by a pair of *j-level links.*

Upper levels. For each j, $\lceil c/2 \rceil + 1 \leq j \leq c$, the nodes $1,\ 1 \cdot i^{j-1} + 1,\ 2 \cdot i^{j-1} + 1,\ 3 \cdot i^{j-1} + 1, \ldots$ are linked together, leaving at most $i^{j-1} - 1$ nodes in a tail.

The sequence of nodes of a c-level jump list linked together by j-level links is called a *j-level list.* A j-level list has maximal length $\lfloor n/i^{j-1} \rfloor = O(i^{c-j+1})$. Note the difference between the lower and upper levels. In the lower levels, each node is involved in a j-level list, while the upper levels only contain one j-level list each, involving only a few nodes.

Figure 3 shows the structure of a 3-level jump list of size 30.

Note that a c-level jump list of size n requires $O(c \cdot n)$ space.

Again, we require that the elements of a set of items of size n be stored in ascending order in the nodes $1, \ldots, n$ in a c-level jump list of size n. This gives a size- and order-unique representation of dictionaries.

To *search* for an item, start with the topmost list and determine the sequence of at most i^{c-1} nodes which may contain the searched item. Then, for each $j = c-1, c-2, \ldots, 1$ follow a sequence of j-level pointers to determine the position of the searched item in the j-level list until it has been found or j has become 1 and the item has not been detected at its expected place in the 1-level list. Note that for each j, $c - 1 \geq j \geq 1$, searching is restricted to a j-level sublist of length at most i. In this way a successful or unsuccessful search can be performed in $O(c \cdot i) = O(c \cdot n^{1/c})$ time in the worst case.

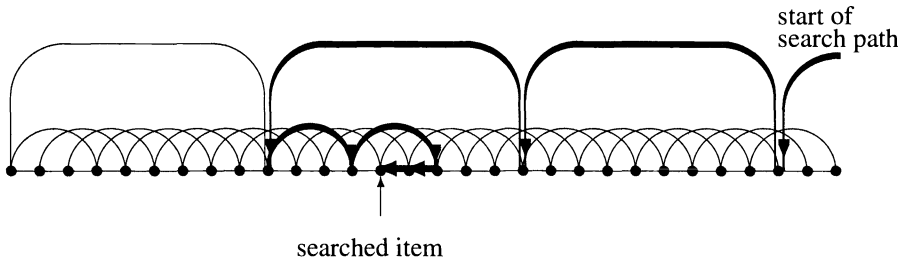


FIG. 4. *Example of a possible search path.*

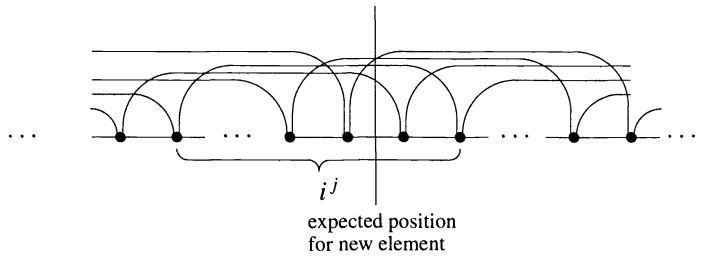


FIG. 5. *j-level lists affected by an insertion.*

In Fig. 4 a possible search path in the 3-level jump list of Fig. 3 has been displayed using arrows and links drawn in bold.

To *insert* an item into a c-level jump list first determine the expected position of the new element by a search as previously explained. Then, the element is inserted into *all* j-level lists, $1 \leq j \leq \lceil c/2 \rceil$. For each j, $1 \leq j \leq \lceil c/2 \rceil$, all j-level links "jumping over" the inserted element will be adjusted; see Fig. 5. That is, an insertion may be considered as the simultaneous insertion of the same new element into i^{j-1} ordered doubly linked lists for all j, $1 \leq j \leq \lceil c/2 \rceil$. This takes time $O(1 + i + i^2 + \cdots + i^{\lceil c/2 \rceil - 1}) = O(i^{\lceil c/2 \rceil - 1})$ altogether. Next, the pointers of all nodes in the upper-level lists to the right of the insertion position have to be shifted by one position to the left. This takes time $O(\sum_{j=\lceil c/2 \rceil + 1}^{c} n/i^{j-1}) = O(\sum_{j=\lceil c/2 \rceil + 1}^{c} i^{c-j+1}) = O(i^{\lfloor c/2 \rfloor})$ in the worst case. The total cost is $O(i^{\lceil c/2 \rceil - 1} + i^{\lfloor c/2 \rfloor})$, which splits into two cases, depending on whether c is even or odd. Thus, it takes $O(\sqrt{n})$ time when c is even and $O(n^{(c-1)/2c})$ when c is odd to insert a new element into a c-level jump list of size n, such that a c-level jump list of size $n + 1$ is obtained.

Deletion can be performed in the analog manner at the same cost; we just make an insertion backwards.

Again, the structure can be fully dynamized by simultaneously maintaining a pair of structures without destroying its features. We summarize our discussion by the following theorem.

THEOREM 1. *For each $c \geq 3$ the c-level jump lists yield a size- and order-unique representation of dictionaries requiring space $O(c \cdot n)$. Dictionary operations are supported at the following worst-case costs:*

search: $O(c \cdot n^{1/c})$;

insertion and deletion: $O(\sqrt{n})$ time when c is even and $O(n^{(c-1)/2c})$ when c is odd.

Choosing $c = 3$ in this theorem balances search and update time and leads to a somewhat surprising result in the light of Snyder's [5] lower bound of $\Omega(\sqrt{n})$.

COROLLARY 1. *3-level jump lists are a size- and order-unique representation of dictionaries which allow us to perform all three dictionary operations (search, insert, delete) in time* $O(n^{1/3})$ *in the worst case.*

It should be noted that any odd value of c gives us a violation of the previous lower bound. The relation between this lower bound and our new upper bound is discussed at the end of this paper.

4. A fully dynamic 3-level structure. In this section we will give a second proof of Corollary 1 by designing a fully dynamic 3-level structure. Hence, we will not refer to the global technique of fully dynamizing the semidynamic 3-level jump list. Instead, by an explicit construction, we will convince the reader that it is possible to "beat" Snyder's lower bound. The new structure, simply called the *jump list*, consists of a directed graph in which a binary search tree is contained. As we will show, both the structure and its maintenance are quite simple.

4.1. Data structure. A jump list of size n consists of n nodes $1, \ldots, n$. These nodes are linked together by three types of pointers:

1st level. The nodes $1, \ldots, n$ are linked together in sorted order in a doubly linked list.
2nd level. For each i, $1 \le i \le n - \lfloor n^{1/3} \rfloor$, there is a pointer from node i to node $i + \lfloor i^{1/3} \rfloor$.
3rd level. The nodes $1^3, 2^3, 3^3, \ldots, \lfloor n^{1/3} \rfloor^3$ are linked together by backward pointers, that is, there is a pointer from node i^3 to node $(i-1)^3$ for each i, $1 < i \le \lfloor n^{1/3} \rfloor^3$.

This specification gives a size- and order-unique representation of dictionaries. It is not hard to see that the structure may be viewed as a binary search tree with some additional pointers. The nodes linked together on the 3rd level make up a left path with the first one of them (from the right) as the root. From each node on this path there is a right path of 2nd-level pointers. Finally, from each node on a right path there is a left path of 1st-level pointers.

We explain this in greater detail. The length of the 3rd-level path, that is, the total number of 3rd-level pointers, is $\lfloor n^{1/3} - 1 \rfloor$. The right subtree of the node at position i^3 contains the elements between this node and its parent (at position $(i+1)^3$). The number of elements in this interval (including position i^3) is $3i^2 + 3i + 1$ and each 2nd-level pointer in the interval points i positions forward. Thus, starting from position i^3, there is a path of right pointers consisting of $3i + 3$ 2nd-level pointers, ending at position $(i+1)^3 - 1$. Finally, from each node on the right path there is a left path of length $i - 1$ consisting of 1st-level pointers.

The elements at positions $\lfloor n^{1/3} \rfloor^3 + 1 \ldots n$ make up the right subtree of the root. Following a chain of 2nd-level pointers from the root, we may not end up at the very last node of the doubly linked list. Thus, the rightmost path in the tree may contain a tail of up to $\lceil n^{1/3} \rceil$ nodes.

We call this tree an *embedded binary search tree*. Note the similarity between this structure and a threaded binary search tree [4]. In both cases the unused pointers at the bottom of the tree point to nodes instead of being nil pointers. An example of a jump list and its embedded binary search tree is given in Fig. 6.

The jump list described here requires four pointers per node. It is possible to use them either by specifying which pointers to use on each level or specifying which pointers to use as left and right pointers in the embedded tree. If desired, with some effort the number of pointers per node may be decreased to three.

LEMMA 1. *A search in a jump list (using the embedded binary search tree) requires* $O(n^{1/3})$ *time in the worst case.*

Proof. Each root-to-leaf path down the embedded binary search tree is composed of three parts corresponding to the three levels of the pointer structure. Each part has a length of $O(n^{1/3})$. From this the lemma follows. □

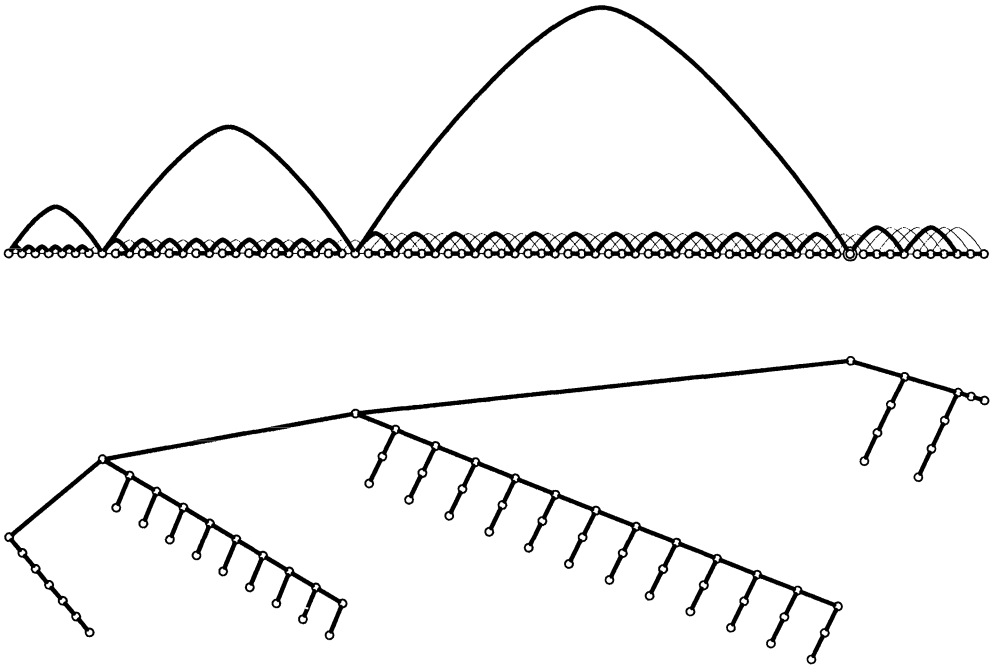


FIG. 6. *A jump list containing 74 elements and the embedded binary search tree.*

4.2. Maintenance algorithms. When inserting/deleting an element, we follow a search path down the embedded binary search tree to find the update position. Then, the element is added to/removed from the 1st-level list. The rest of the structure is adjusted by changing some 2nd-level pointers in the neighborhood of the update position and reconstructing a part (maybe all) of the 3rd-level list.

In detail, during an *insertion* the data structure is modified in the following way:

1. The new element is inserted into the 1st-level list (at position p).
2. A 2nd-level pointer going out from p is created and each 2nd-level pointer going out from a node to the left of p and ending to the right of p is shifted one position to the left, that is, it now points to the left neighbor of its previous target node.
3. The 3rd-level list is reconstructed such that each node in this list to the right of p is replaced by its predecessor. That is, the 3rd-level pointer jumping over the insertion position has to be "shortened" by one, and as a result of this all other 3rd-level pointers to the right have to be shifted by one position to the left. If the tail becomes too long, eventually one new 3rd-level pointer has to be created also. At each position where the 3rd-level list is shifted, we also have to change the 2nd-level pointer.

The *deletion* algorithm works analogously by performing an insertion backwards.

Example. If we insert a new element between positions 40 and 41 into the structure in Fig. 6, we have to change the pointers that are marked by bold lines in Fig. 7. The new pointers that occur are drawn downwards.

LEMMA 2. *The cost of an update is* $O(n^{1/3})$ *in the worst case.*

Proof. From the description of the maintenance algorithms it follows that after locating the update position, $O(1)$ pointer changes are made on the 1st level and $O(n^{1/3})$ are made on the other two levels. Each pointer change requires $O(1)$ time, except when the length of the 3rd-level path is increased. In that case a search is required to locate the last node. The total cost of this is $O(n^{1/3})$, which completes the proof. □

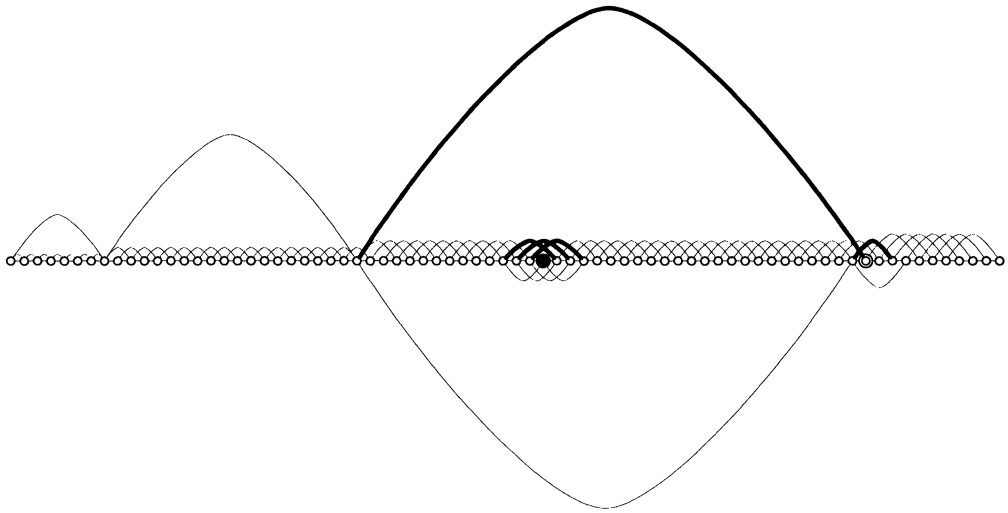


FIG. 7. *Performing an insertion at position* 41 *in the data structure of Fig.* 6.

Altogether we have the following theorem.

THEOREM 2. *Jump lists are a representation of dictionaries with the following properties:*

- *they are size- and order-unique;*
- *a dictionary of size n requires* $O(n)$ *space;*
- *the cost of performing a search, insert, or delete operation in a jump list which stores a dictionary of size n is* $O(n^{1/3})$.

Note that the operation of exchanging elements between nodes has not been used in our upper bound construction. We only use creation or deletion of nodes and pointer changes. There are also no "hidden" costs required to locate nodes where pointer changes should be made. All these locations can be found within the time bound of $O(n^{1/3})$.

5. A lower bound on unique representation. We give a lower bound on maintaining size- and order-unique representations of dictionaries. Recall that we presuppose graphs with bounded out-degree ($= k$) only. We assume that the size of the graph is unchanged and, as an update, we regard the operation of deleting one element and inserting another. That is, after an update the same graph must occur; only the stored elements may change. Also recall that there is a one-to-one mapping between nodes and elements. Thus, when arguing about the graph, we may argue about elements connected by edges instead of nodes.

We say that element y is a *parent* of element x if there is an edge from y to x. The set of all x's parents is called the *parent-set* of x.

We assume that the elements are taken from an ordered universe, and by the *rank* of an element x, denoted $\text{rank}(x)$, we mean the number of elements smaller than x in the stored set plus one. Let x, y, and z be three elements such that $\text{rank}(x) < \text{rank}(y) < \text{rank}(z)$. An edge from x to z has a *length* of $\text{rank}(z) - \text{rank}(x)$ and *covers* the element y; we also say that element z covers y. An edge from z to x has a negative length (and also covers y). The *incoming pattern* of an element is given by the lengths of its incoming edges. Since the graph is unique, the incoming pattern of an element x is exactly determined by the rank of x. Thus, each rank is associated with a specific incoming pattern. We say that a rank r is *critical* if the pattern associated with r differs from the pattern associated with $r + 1$. An element that has a critical rank must change its incoming pattern when an update causes its rank to increase by one.

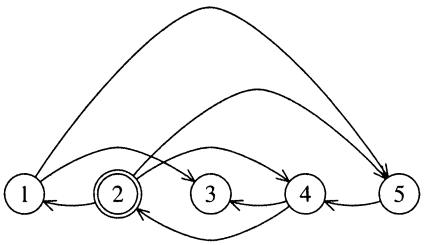


FIG. 8. *A graph.*

Note the difference between parent-sets and incoming pattern. The parent-set of x tells *which elements* are the parents of x, while the incoming pattern of x deals with *differences in ranks* between x and its parents.

In a graph G, the longest distance from the root to an element is denoted by D_G, the largest number of nodes covering one node is denoted by C_G, and the number of critical ranks is denoted by P_G.

Example. In Fig. 8 a graph G with $k = 3$ is shown. The nodes are labeled with the ranks of the contained elements. The root is 2. The parent-set of element 3 is the set of elements 1 and 4. Elements 3 and 4 have the same incoming pattern (edges of lengths 2 and -1). This implies that rank 3 is not critical, while all other ranks (except the last rank) are critical. All elements can be reached from the root by traversing at most two edges, thus $D_G = 2$. Element 3 is covered by four edges, (1→5, 2→4, 2→5, and 4→2). However, two of the edges lead to the same element, so the number of elements covering 3 is three (2, 4, and 5). This is the maximum number of covering elements, thus $C_G = 3$.

We start by showing some properties of a graph in Lemmas 3 and 4. Next, in Lemma 5 we compute the cost of structural changes. In Lemmas 6–9 we show that "bad" updates will enforce a high restructuring cost. Finally, the lower bound is given in Theorem 3.

LEMMA 3. *If* $0 \neq 3D_GC_G < n$, *then in each set of* $3D_GC_G$ *consecutive elements not containing the root there is at least one element with a critical rank.*

Proof. If there is no critical rank all elements must have the same incoming pattern. We prove the lemma by showing that this may not occur.

First, assume that to each element there is an edge of length L, $L > C_G + 1$. Then, the first element in the set is covered by at least $C_G + 1$ elements. Similarly, if to each element there is an edge of length L, $L < -(C_G + 1)$, the last element in the set is covered by at least $C_G + 1$ elements. In both cases we get a contradiction of the definition of C_G.

Second, assume that to each element there is no edge longer than $C_G + 1$. Let p be the element with the median rank in the set. To reach p from outside the set we must pass a distance of $1.5D_GC_G$. Doing this by following edges of length at most $C_G + 1$, we would have to follow more than $1.5D_GC_G/(C_G + 1) > D_G$ edges. We get a contradiction of the definition of D_G.

Thus, our assumption that all elements have the same incoming pattern leads to a contradiction. This completes the proof. □

LEMMA 4. *If* $0 \neq 3D_GC_G < n/2$ *then* $D_G \cdot C_G \cdot P_G = \Omega(n)$.

Proof. From Lemma 3 we know that all (but one) sets of $3D_GC_G$ consecutive elements contain one element with a critical rank. This gives

$$P_G \geq \frac{n}{3D_GC_G} - 1 = \Omega\left(\frac{n}{D_GC_G}\right). \tag{1}$$

The last equality follows, since $3D_GC_G < n/2$. □

LEMMA 5. *Changing the parent-sets of m elements requires a cost of* $\Omega(m)$.

Proof. We prove the lemma by showing that each restructuring operation changes the parent-set of $O(k) = O(1)$ elements. From this fact it follows that $\Omega(m)$ restructuring operations are required to change the parent-sets of m elements. Recall that from the model of computation we have the following possible operations:

- Create a node. This operation does not affect the parent-set of any element. (If we allow the new node to have k outgoing edges, the number of changed parent-sets would be k).
- Remove a node. This operation affects the parent-set of at most $k + 1$ elements: the element that was in the node and the elements reached from that node.
- Change/add/remove one outgoing edge from a node (pointer change). This operation changes the parent-set of at most two elements: one loses a parent and one gets a new parent.
- Exchange elements between two nodes. This operation changes the parent-set of at most $2k + 2$ elements: the two exchanged elements and their children.

Thus, each operation changes the parent-set of $O(1)$ elements, which completes the proof. □

LEMMA 6. *Let x be an element with critical rank. Then, x cannot have the same parent-set as before, after the following update*:

1. *delete the largest element*;
2. *insert a new smallest element.*

Proof. Let X denote the set of $n - 1$ (consecutive) elements that is stored in the graph both before and after the update.

Assume that the parent-set of x is the same before and after the update. This implies that neither the deleted element nor the inserted element can be a parent of x. Thus, the parent-set only consists of elements in X. (Note that x also belongs to X.)

The update described in the lemma will cause the rank of each element in X to increase by one, and hence the difference in rank between any two elements in X will be the same after the update as before the update. Thus, if x has the same parent-set before and after the update, each incoming edge will have the same length, and thus the incoming pattern of x will be the same. We get a contradiction with the definition of a critical rank, which completes the proof. □

LEMMA 7. *There is an update which requires a cost of* $\Omega(P_G)$.

Proof. To prove the lemma we delete the largest element and insert a new smallest element. This implies that the rank of each element (except the first one) will increase by one. Thus, from Lemma 6 it follows that each element that had a critical rank before the update must change its parent-set. This together with Lemma 5 completes the proof. □

LEMMA 8. *Let x and y be two elements,* $\text{rank}(x) < \text{rank}(y)$, *such that x is y's smallest parent. Then, the parent-set of y must be different after the following update*:

1. *delete the smallest element*;
2. *insert a new element between x and y (or anywhere if x was the smallest element).*

Proof. The lemma is trivially true if x was the smallest element and therefore deleted.

Otherwise, we prove the lemma by showing that after the described update, there cannot be an edge from x to y.

After the described update the rank of y is unchanged, and thus y has the same incoming pattern. This implies that the difference in rank between y and its smallest parent (i.e., the length of y's longest incoming edge) must be the same as before the update. However, after the update the rank of x is decreased by one. Thus, if there would be an edge from x to y after the update, the length of the longest incoming edge of y would change. We get a contradiction, which completes the proof. □

Note that there is a symmetric version of Lemma 8 for y's largest parent (with a larger rank). Because of the without-loss-of-generality- (w.l.o.g.-) assumption below, this symmetric lemma is not needed.

LEMMA 9. *There is an update which requires a cost of* $\Omega(C_G)$.

Proof. We assume w.l.o.g. that there is an element p which is covered by $\Theta(C_G)$ elements $q_1, q_2, \ldots$ with larger ranks. To prove the lemma we delete the smallest element and insert a new element which becomes the immediate successor of p in the ordered set. This will have the effect that the situation described in Lemma 8 will occur for each element q_i. This fact together with Lemma 5 completes the proof. □

THEOREM 3. *For any size- and order-unique graph representation of a dictionary there is a dictionary operation which requires* $\Omega(n^{1/3})$ *time.*

Proof. If $C_G = 0$ the graph G must be a linked list with $D_G = \Omega(n)$. Hence, w.l.o.g. we may assume that $C_G > 0$. From Lemma 4 it follows that at least one of the three D_G, C_G, and P_G's has to be $\Omega(n^{1/3})$. Since the search cost is $\Omega(D_G)$ and the update cost is $\Omega(\text{Max}(C_G, P_G))$, either a search or an update has to require $\Omega(n^{1/3})$ time. □

Note that if an update involves a search for the update position, then the update requires $\Omega(n^{1/3})$ time.

Note also that our lower bound for size-uniqueness may be transformed to be valid for set-uniqueness by using Ramsey's theorem [2] in the same way as Sundar and Tarjan [6].

6. Comments and open problems. We feel that our findings point to a fundamental fact in the theory of data structures. In particular, in the presence of earlier lower bounds for unique representations [5], [6], stating that $\Theta(\sqrt{n})$ time is required per dictionary operation, the new upper bound presented above might seem surprising.

The lower bound given by Snyder [5] is based on the implicit assumption that changing the status of an element from being stored in a leaf of a binary tree to being stored in an internal node (or vice versa) requires $\Omega(1)$ time. As we have shown, this change can be achieved without explicitly performing any operation at the node if the binary search tree is embedded in a directed graph appropriately.

The lower bound by Sundar and Tarjan, stating that $\Theta(\sqrt{n})$ CONS operations are required per update, is roughly based on the following argument: By choosing an "adversary" sequence of updates in a uniquely represented binary search tree, at each update we can enforce the occurrence of $\Omega(\sqrt{n})$ new subtrees, which have never existed before. From the assumption that each occurring subtree has to be constructed at some moment, and the construction requires $\Omega(1)$ time per subtree, a lower bound of $\Omega(\sqrt{n})$ time per update follows.

The first argument, in which $\Omega(\sqrt{n})$ new trees may occur per update, is true for all uniquely represented dictionaries by binary search trees and also for the "embedded" trees presented here. However, all new subtrees may not need explicit construction. The assumption that $\Omega(1)$ cost per new subtree is needed seems reasonable if CONS is the only primitive to manipulate trees. But it does not hold if search trees are embedded in directed graphs that can be updated by pointer changes.

It is intuitively clear that CONS operations may be implemented by creating or removing nodes and changing the pointers of a set of nodes in a directed graph. Hence, it is possible to obtain a size- and order-unique representation of dictionaries which is based on the ideas of Sundar and Tarjan but uses pointer changes as primitives. This leads to the notion of *shared search trees* [3]. Shared search trees are a size- and order-unique representation of dictionaries requiring space $O(n \log n)$. For a shared search tree storing a dictionary of size n, a search operation can be performed in time $O(\log n)$ and updates can be performed in time $O(\sqrt{n})$. The shared search tree is a directed graph with nodes of unbounded degree $\Theta(\log n)$. It is not known whether there exists a set- and order-unique representation of dictionaries such that

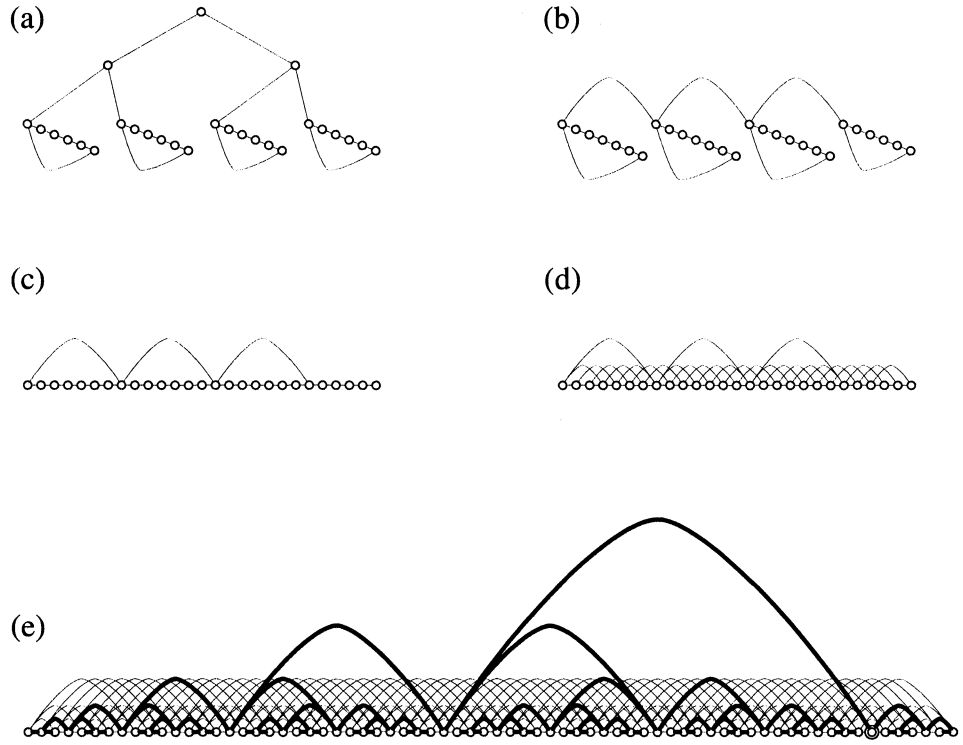


FIG. 9. *Evolution of structures.* (a) *The jelly-fish structure of Snyder* [5]. *Each "tentacle" consists of a circular list.* (b) *The same structure with the tree-shaped "body" replaced by a list.* (c) *The edges connecting the top and bottom of each tentacle is replaced by a forward-link. We achieve a* 2-level *jump list.* (d) *The* 2*nd level links are added and we have the* 3*-level jump list.* (e) *Using* $\Theta(\frac{\log n}{2})$ *levels we obtain the shared search tree, a structure similar to the one described by Sundar and Tarjan* [6] *with a search cost of* $O(\log n)$ *and an update cost of* $O(\sqrt{n})$. *The root is marked by a double circle.*

the underlying graph has bounded node-degree, a search operation can be performed in time $O(\log n)$, and updates take time $O(\sqrt{n})$ or less in the worst case using pointer changes as primitive operations.

There is a uniform way to evolve all the structures discussed in this paper, which is illustrated in Fig. 9.

Our results also raise the following more general problem: All known classes of (balanced) search trees are not set-unique representations of dictionaries, because in general the same set of items may be represented by exponentially many different trees. Thus, there is a large gap between unique representations and commonly used classes of trees to represent dictionaries. The question is how efficiently can search and update algorithms operate if a dictionary of size n is represented by $f(n)$ different structures. In this paper we have dealt with the case $f(n) = O(1)$. The classical theory of (balanced) binary search trees is based on the assumption that $f(n)$ is exponential. Nothing is known for any function f strictly in between these two extremes.

Acknowledgments. We thank Bengt Nilsson, Sven Schuierer, and Svante Carlsson for comments and discussions. The comments by the anonymous referees have led to improvements to both the content and the presentation.

REFERENCES

[1] C. R. ARAGON AND R. G. SEIDEL, *Randomized search trees*, in Proc. 30th IEEE Symposium on Foundations of Computer Science, Research Triangle Park, NC, 1989, pp. 540–545.

[2] R. L. GRAHAM, B. L. ROTSCHILD, AND J. SPENCER, *Ramsey Theory*, John Wiley, 1980.

[3] T. OTTMANN, *Trees—a personal view*, in New Results and New Trends in Computer Science, Lecture Notes in Computer Science 555, Springer-Verlag, Graz, 1991.

[4] A. J. PERLIS AND C. THORNTON, *Symbol manipulation by threaded lists*, Comm. Assoc. Comput. Mach., 1 (1960), pp. 195–204.

[5] L. SNYDER, *On uniquely representable data structures*, in Proc. 18th IEEE Symposium on Foundations of Computer Science, 1977, pp. 142–146.

[6] R. SUNDAR AND R. E. TARJAN, *Unique binary search tree representations and equality-testing of sets and sequences*, in Proc. 22nd ACM Symposium on Theory of Computing, Baltimore, MD, 1990, pp. 18–25.

SIAM J. COMPUT.
Vol. 24, No. 5, pp. 1104–1121, October 1995

011

SCHEDULING JOBS WITH TEMPORAL DISTANCE CONSTRAINTS*

CHING-CHIH HAN†, KWEI-JAY LIN‡, AND JANE W.-S. LIU§

Abstract. The job scheduling problems for real-time jobs with temporal distance constraints (JSD) are presented. In JSD, the start times of two related jobs must be within a given distance. The general JSD problem is NP-hard. We define the multilevel unit-time JSD (MUJSD) problem for systems with m chains of unit-time jobs in which neighboring jobs in each chain must be scheduled within c time units. We present an $O(n^2)$-time algorithm, where n is the total number of jobs in the system, and also an $O(m^2c^2)$-time algorithm. Some other variations of the JSD problems are also investigated.

Key words. deadline, job scheduling, precedence constraint, real-time systems, relative timing constraint, temporal distance

AMS subject classification. 68Q25

1. Introduction.

In job scheduling problems, many timing requirements are defined as independent conditions with absolute values. For example, each job has predefined *ready time* and *deadline*. In some problems, jobs may also have *precedence* constraints, i.e., some jobs must be finished before others can start. Job scheduling algorithms for real-time systems are then designed to find *feasible* schedules in which all ready times, deadlines, and precedence constraints are satisfied.

In many real-time applications, however, jobs must satisfy relative timing constraints. For example, in a chemical process control system, one element must (or cannot) be added into a processing unit within a certain time after another element has been put in. In a painting process, a new coat of painting usually cannot be put on until a certain time after the previous coat is completed. In all of these problems, the applications impose some relative distances between the executions of related jobs. We call such constraints *temporal distance constraints* [8], [9] and *separation constraints* [10].

Jobs with a temporal distance constraint must be executed inside the given time interval of each other, while jobs with a separation constraint must be executed with a minimum interval between them. In other words, a job has a relative ready time or deadline depending on when its predecessor was executed. Given a set of jobs $\mathbf{J} = \{J_1, J_2, \ldots, J_n\}$, in which each job J_i has execution time e_i, ready time r_i, and deadline d_i, $1 \leq i \leq n$, the *job scheduling with distance constraint* (JSD) problem is to find a start time function f such that for $1 \leq i, j \leq n$, and $i \neq j$,

(1) $f(J_i) \geq r_i$,
(2) $f(J_i) + e_i \leq d_i$, and
(3) $|f(J_i) - f(J_j)| \leq w(J_i, J_j)$.

In the third condition, $w(J_i, J_j)$ is the distance constraint between J_i and J_j. If there is no distance constraint between J_i and J_j then $w(J_i, J_j) = \infty$. For the *job scheduling with*

*Received by the editors August 12, 1991; accepted for publication (in revised form) May 23, 1994. This work was supported in part by Office of Naval Research grant N00014-89-J-1181, National Science Foundation grant CCR-89-11773, and National Aeronautics and Space Administration grant NAG-1-613.

†Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122 (cchan@eecs.umich.edu). This work was done while this author was a Ph.D. student at the Department of Computer Science, University of Illinois at Urbana-Champaign.

‡Department of Electrical and Computer Engineering, University of California, Irvine, CA 92717 (klin@ece.uci.edu). This work was done while this author was an assistant professor at the Department of Computer Science, University of Illinois at Urbana-Champaign.

§Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801 (janeliu@cs.uiuc.edu).

separation constraint (JSS) problem, the condition should be changed to

(3′) $|f(J_i) - f(J_j)| \geq w(J_i, J_j)$.

In this paper, we study the JSD problems with various job models. We concentrate on the single processor scheduling problems; the multiprocessor scheduling problem will be briefly discussed at the end of this paper.

The rest of the paper is organized as follows. Next we discuss some related work. In §2, we discuss the JSD problem in general and also define the multilevel unit-time JSD (MUJSD) problem, in which there are multiple chains of unit-time jobs all with the same distance constraint. An $O(n^2)$-time algorithm SMD for the MUJSD problem of n jobs is presented in §3. In §4, a polynomial-time algorithm PMD for the MUJSD problem is also presented. Section 5 discusses some extensions of the MUJSD problem. We conclude the paper in §6.

1.1. Related work. In job scheduling problems, jobs must meet their timing constraints, such as ready times, deadlines, and precedence constraints. The general problem of scheduling jobs with different ready times and deadlines is known to be NP-complete [4], [5]. However, if all jobs have the same execution time, the problem can be solved in polynomial time even with precedence constraints [14]. There are many excellent survey papers on the problems of deterministic job scheduling [2], [6], [11], [12].

In most of the previous work on job scheduling, job deadlines have fixed values. Very little work was done on jobs with relative deadlines. A problem related to the JSD problem is the *linear array problem* (LAP) [13]. Given a graph G in which each edge is labeled with an integer value, LAP tries to arrange the vertices in a linear sequence. If two vertices are adjacent in G, their distance in the sequence must not be larger than their linkage value in G. It has been shown that the general LAP and some of its special cases are NP-complete [3], [13]. One of the NP-complete LAP cases is the *bandwidth minimization problem* (BMP), where all edges are labeled with the same integer k. The BMP remains NP-complete for rooted directed trees where the maximum in-degree is one and the maximum out-degree is at most two [3], [5].

2. Scheduling jobs with distance constraints. As we already defined, distance constraints are defined for all i and j such that $|f(J_i) - f(J_j)| \leq w(J_i, J_j)$. In this paper, we consider only the nonpreemptive JSD problem. The preemptive JSD problem is discussed in [9]. We first consider the single-processor systems. The multiprocessor JSD problem will be discussed in §5.

If jobs have different execution times, it is not hard to construct a special case with only one distance constraint and show that the problem is NP-complete [7], [8]. We therefore investigate only the nonpreemptive JSD problems in which all jobs have the same execution time and the same ready time here. We call this problem the unit-time JSD (UJSD) problem. Without loss of generality, we assume that all jobs require one unit of execution time and all ready times are equal to 0. Also, we first assume that deadlines and distance constraints are all integers. This assumption will be relaxed in §5 to include fractional number deadlines and distance constraints. If all jobs have deadlines greater than or equal to the total number of jobs, the UJSD problem is equivalent to LAP. This is obvious since we can use vertices to represent the jobs and edges with labels to represent the distance constraints between jobs. Thus, we know that the UJSD problem is also NP-complete.

2.1. The MUJSD problem. In the following discussion, we restrict our attention to a special class of the UJSD problem in which the graphs have a directed multichain tree structure [1]. We will show that if all distance constraints are the same, the problem can be solved in polynomial time. However, the problem becomes NP-complete if different distance constraints are allowed for different pairs of jobs.

In an MUJSD system, the job set is divided into chains of jobs. In each chain, only the first job has a deadline. The other jobs have a constant distance constraint with their immediate predecessors. Formally, there are two types of jobs in an MUJSD system: *head* job set **H** and *tail* job set **T**. Each job H_i in $\mathbf{H} = \{H_1, H_2, \ldots, H_m\}$ has a deadline d_i. Jobs in **T** are grouped into m subsets $T_1, T_2, \ldots, T_m$, where $T_i = \{J_{i1}, J_{i2}, \ldots, J_{ik_i}\}$, and $k_i \geq 0$. T_i is the set of tail jobs after the head job H_i, while J_{ij} is the jth tail job of H_i. We define $J_{i0} = H_i$.

To satisfy the temporal distance constraint, job J_{ij} must be started within c time units *after* its immediate *predecessor* $J_{i,j-1}$ is started. That is, there is a precedence constraint from $J_{i,j-1}$ to J_{ij} and the distance constraint between these two jobs is a constant c. If c is larger than or equal to the number of head jobs m, it is easy to find a schedule which can satisfy the constraint. We thus assume c is less than m. The total number of jobs n in the MUJSD system is $m + \sum_{i=1}^{m} k_i$.

The problem can be represented by a directed multichain tree structure as follows. Each job is defined as a vertex in the tree. Each job sequence, $J_{i0}, J_{i1}, \ldots, J_{ik_i}$, is defined as a chain in the tree with a directed edge labeled c between all adjacent vertices in the chain. A dummy vertex R is defined as the root of the tree. We also define a directed edge from R to each H_i in **H** with a label of the deadline d_i (Fig. 2.1). Thus the structure has multiple levels and all edges except those on the first level have label c.

We have designed two different algorithms for the MUJSD problem. The first algorithm SMD has a time complexity of $O(n^2)$, where n is the number of jobs in the system. The complexity is a polynomial function of the output size, since SMD will find the start times for all of the n jobs. However, it is a pseudopolynomial function of the input size since only the m, c, and k_i's, need to be specified in the input. We thus present another algorithm PMD which is an $O(m^2c^2)$ polynomial-time algorithm of the input size.

3. A pseudopolynomial time algorithm for MUJSD. Before we present the SMD algorithm for the MUJSD problem, we first show some properties of the MUJSD schedules. These properties will motivate the design of the SMD algorithm.

3.1. Properties of the MUJSD schedules. In our discussion, we use *time slot*, or just *slot*, t to refer to the unit-time interval $[t-1, t]$. Therefore, if we say job J is scheduled at slot t or slot t is assigned to (occupied by) J in a schedule f, it means that J will be finished by t or $f(J) = t - 1$. We define the *virtual deadline* d_{ij} of job J_{ij} as $d_{ij} = d_i + j \cdot c$ for $1 \leq i \leq m$ and $0 \leq j \leq k_i$. Thus, $d_{i0} = d_i$. In a feasible schedule, job J_{ij} must be scheduled at a slot no later than d_{ij}. However, scheduling J_{ij} at slot d_{ij} may not be sufficient to satisfy the distance constraint for J_{ij} since its predecessor $J_{i,j-1}$ may have been scheduled at a slot earlier than $d_{ij} - c$. In other words, the virtual deadline of a job is only an "upper bound" of its completion time.

For any time t, we define $h(t)$ as the number of jobs with virtual deadlines less than or equal to t. Clearly, if there are more jobs to be executed than the time slots available, there cannot be any feasible schedule for the system. In other words, for an MUJSD system with n jobs, if there exists a t, $1 \leq t < n$, such that $h(t) > t$, then there is no feasible schedule for the MUJSD system.

In SMD, the final schedule is constructed by adding one job at a time to an initially empty schedule. In what follows, when we say a job J is scheduled in a partial schedule f, we mean that $f(J)$ is defined. We call a partial schedule *well formed* if, for all jobs scheduled in the partial schedule, all their predecessors must have been scheduled in the partial schedule and all the timing constraints on them are satisfied.

DEFINITION. *A schedule f is called a* well-formed partial schedule *if, for all J_{il} scheduled in f, J_{ip}, $0 \leq p \leq l-1$, is also scheduled in f, and all constraints for J_{il} are satisfied.*

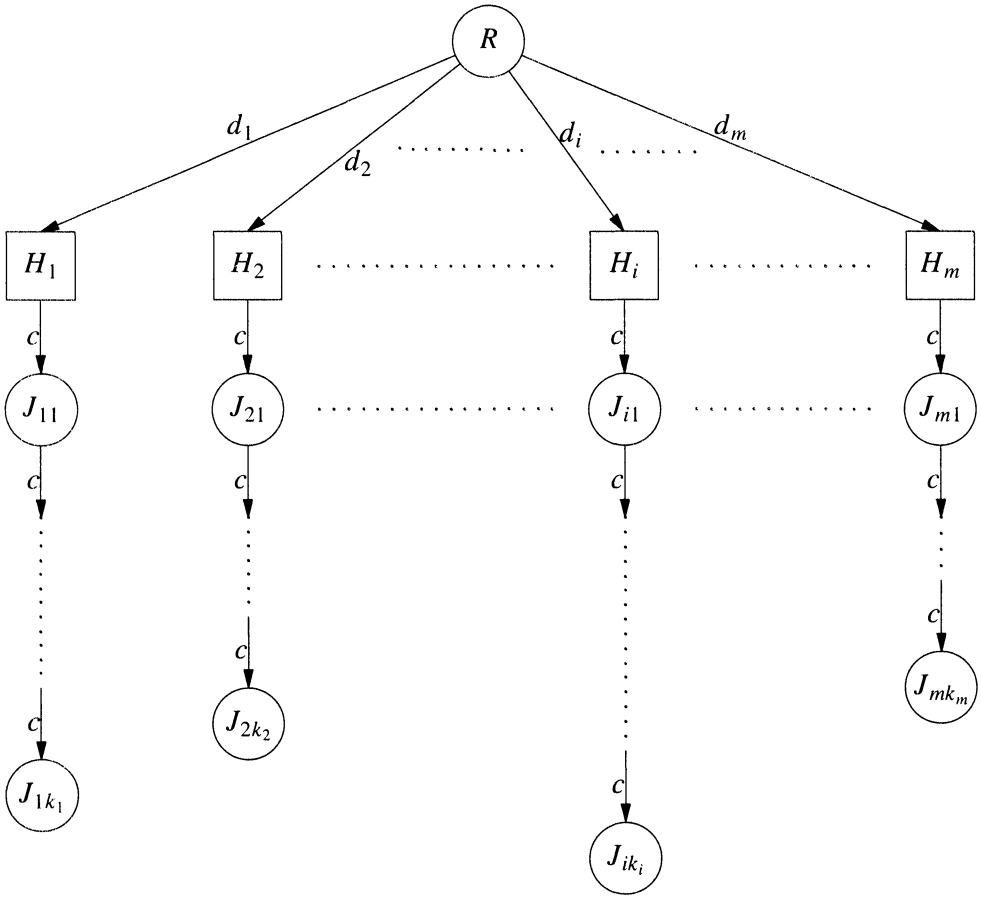


FIG. 2.1. *The m-chain tree structure of the MUJSD problem.*

In the following discussion, we sometimes omit the words well formed if it is obvious in the context. We also say that job J_{ij} can be *feasibly scheduled* into a well-formed partial schedule f if J_{ij} can be added to f to get another well-formed partial schedule which satisfies all of the timing constraints.

DEFINITION. *A job J_{il} is* reschedulable *in a partial schedule f if J_{il} has been scheduled in f, which satisfies either condition R1 or R2:*

R1. $J_{i,l+1}$, J_{il}'s *immediate* successor, *is scheduled in f and $f(J_{i,l+1}) < f(J_{il}) + c$;*

R2. *none of J_{il}'s successors has been scheduled in f.*

Let us now consider the condition that a job can be added to a partial schedule. To add a job into a partial schedule, we need to find an empty slot for it. If a head job H_i is to be scheduled into a partial schedule f and if there is an empty slot before d_i, then we can simply schedule H_i in the empty slot closest to and before d_i.

However, if the job to be added is a tail job, the condition is not so simple. This is because for a tail job, both precedence and distance constraints must be satisfied. To satisfy the precedence constraint, we need to find an empty slot after the job's predecessor. To satisfy the distance constraint, the distance between the empty slot and the job's predecessor must be less than or equal to c. If there is any empty slot which can satisfy both the precedence and distance constraints, we can simply schedule the job at the latest such empty slot. However,

if there is an empty slot but it is before the job's predecessor, we need to relocate the empty slot (or the predecessor) so that the empty slot can be used by the job without violating the precedence and the distance constraints.

In the following lemma we will see that if a job is reschedulable in a partial schedule and if there is an empty slot in front of the job's currently scheduled slot, the job's slot can be made empty by rescheduling the jobs at some earlier time slots. The proof of the lemma uses the fact that if a job is reschedulable, it has the freedom to move forward in the partial schedule. The exact final location of the job depends on what kind of jobs are in front of it in the partial schedule. In all cases, we can show that the partial schedule after the move still satisfies all of the constraints.

LEMMA 3.1. *Given a partial schedule f and a reschedulable job J_{il} currently scheduled at slot Q, if P is the closest empty slot before slot Q, we can find a new partial schedule g by rescheduling only the jobs scheduled between P and Q, so that slot Q is empty in g and $g(J_{il}) \geq Q - c$, where c is the distance constraint.*

Proof. We prove the lemma by induction on $D = Q - P$, the distance between the empty slot and J_{il}'s scheduled slot in f. Since J_{il} is reschedulable, either R1 or R2 is true for J_{il}.

Induction base. If $D = 1$, i.e., the empty slot is immediately before Q, then we can move J_{il} to slot P to get g. If J_{il} is a tail job, the move will not violate the distance constraint between J_{il} and its predecessor. If J_{il} is a head job, the move will not violate its deadline constraint. If R1 is true, i.e., $f(J_{i,l+1}) < f(J_{il}) + c$, then $g(J_{i,l+1}) \leq g(J_{il}) + c$; the distance constraint is still satisfied. If R2 is true, i.e., $f(J_{i,l+1})$ is undefined, no distance constraint between J_{il} and its successors can be violated in g. Moreover, $g(J_{il}) = f(J_{il}) - 1 \geq f(J_{il}) - c = Q - c$.

Induction hypothesis. Assume the lemma is true for any partial schedule f and a reschedulable job J_{il} in f with $D < u$.

Induction step. We prove that the lemma holds for any partial schedule f and a reschedulable job J_{il} in f with $D = u$. We need to prove two cases.

Case 1. R1 is true for J_{il} (i.e., $f(J_{i,l+1}) < f(J_{il}) + c$).

Suppose slot R is assigned to $J_{i,l+1}$. We have $R - Q < c$. Let us inspect slots $Q - 1$, $Q - 2, \ldots, R - c$ one at a time. We will eventually find one of the following:

(1) the empty slot P,
(2) the slot s assigned to $J_{i,l-1}$, or
(3) a slot s' assigned to a job which is reschedulable in f.

If slots P and s are not found first, we will definitely find an s'. This is because if slot $R - c$ is not empty or assigned to $J_{i,l-1}$, then the job scheduled at slot $R - c$ must either have its immediate successor scheduled at a slot earlier than slot R (since slot R is now occupied by $J_{i,l+1}$) or have no successor scheduled, which means that it must be a reschedulable job in f. Therefore, at least slot $R - c$ satisfies one of the above three conditions. We now show how to find g in all these situations.

(1) If we first find the empty slot P, we can simply reschedule J_{il} at P and leave slot Q empty to get a partial schedule g.
(2) If we first find the slot s, by induction hypothesis we can get a new partial schedule f' such that slot s is empty and $f'(J_{i,l-1}) \geq s - c$ (since $f(J_{il}) - f(J_{i,l-1}) < c$, i.e., $J_{i,l-1}$ is reschedulable in f and $s - P < u$). Then, we can reschedule J_{il} at slot s and leave slot Q empty to get a well-formed partial schedule g.
(3) If we first find a slot s', which is assigned to a reschedulable job, again by induction hypothesis we can find a new well-formed partial schedule f' such that slot s' is empty. Then, we can reschedule J_{il} at the new empty slot s' and leave slot Q empty to get a well-formed partial schedule g.

In all situations $g(J_{il}) \geq f(J_{il}) - c = Q - c$.

Case 2. R2 is true for J_{il} (i.e., none of J_{il}'s successors has been scheduled).

In this case, we will inspect slots $Q-1, Q-2, \ldots, Q-c$ one at a time. If, before we reach slot $Q-c$, we find one of the slots P, s, or s' as defined in Case 1, the lemma can be proved as in Case 1. If we cannot find any such slot before we inspect slot $Q-c$, then the lemma can be proved as follows:

(1) If J_{il} is a head job, then slot $Q-c$ is either empty or occupied by a reschedulable job, since the job scheduled at slot $Q-c$ must have its immediate successor scheduled at a slot earlier than slot Q or have no successor scheduled. The lemma can be proved as in Case 1.

(2) If J_{il} is a tail job, then the job scheduled at slot $Q-c$ must be $J_{i,l-1}$. If we remove J_{il} from f to get a new partial schedule f', then $J_{i,l-1}$ is a reschedulable job in f'. By induction hypothesis, we can get a new partial schedule g' from f' such that slot $Q-c$ is empty in g', and $g'(J_{i,l-1}) \geq f(J_{i,l-1}) - c$. Then, we can reschedule J_{il} at slot $Q-c$ to get a well-formed partial schedule g without violating the distance constraint between J_{il} and $J_{i,l-1}$.

In both Cases 1 and 2 the only jobs rescheduled are those jobs that were scheduled between P and Q. The lemma is true by induction. □

From Lemma 3.1, we can derive a sufficient condition that a tail job can be feasibly scheduled into a partial schedule. This is shown in the following lemma.

LEMMA 3.2. *In an MUJSD system, suppose a tail job J_{il} is not scheduled in a partial schedule f but all its predecessors have been scheduled. If there is any empty slot before $f(J_{i,l-1}) + c + 1$, then we can always find a new well-formed partial schedule in which J_{il} is scheduled.*

Proof. If there is any empty slot between $f(J_{i,l-1}) + 1$ and $f(J_{i,l-1}) + c + 1$, job J_{il} can be scheduled at the empty slot closest to and before $f(J_{i,l-1}) + c + 1$. If there are empty slots only before $f(J_{i,l-1})$, from Lemma 3.1, we can always find another partial schedule f' so that slot $f(J_{i,l-1}) + 1$ is empty and $f'(J_{i,l-1}) \geq f(J_{i,l-1}) - c$. J_{il} can then be scheduled at $f(J_{i,l-1})$. It is easy to see that the new partial schedule is still well formed in both cases. □

3.2. The SMD algorithm. We have shown that, for a job J_{il} yet to be scheduled into a partial schedule f, if all its predecessors have been scheduled and there is an empty slot before the time that J_{il} must be finished, then J_{il} can always be feasibly scheduled into f. With this property, we now present the scheduling algorithm SMD for the MUJSD problem.

ALGORITHM SMD

Step 1. Sort the jobs into $S_1, S_2, \ldots, S_n$ with nonincreasing number of successors.

Step 2. For i from 1 to n do {

```
        if S_i is a head job H_j then SCHED(i, d_j, 0)
        else { suppose S_i's predecessor is scheduled at slot s;
               SCHED(i, s + c, 0); }
    }

procedure SCHED(i, t, r);
{
  if t = 0 then output "unschedulable" and stop
  else if slot t is empty then f(S_i) = t - 1  /* schedule S_i at slot t */
       else {
            suppose slot t is now assigned to S_k;
            if (S_k is the predecessor of S_i) or (r = 1 and S_k is reschedulable)
            then { f(S_i) = t - 1;  /* schedule S_i at slot t */
```

```
            SCHED(k, t − 1, 1); }  /* reschedule S_k */
        else SCHED(i, t − 1, r);
      }
}
```

In SMD, we schedule one job at a time. Among all jobs remaining to be scheduled, we always pick the job with the largest number of successors to schedule next. If the job is a head job we schedule it at the empty slot, if any, closest to and before the job's deadline. If the job is a tail job with its predecessor scheduled at slot s, and if there exists any empty slot between time s and time $s + c$, we can simply schedule the job at the empty slot closest to time $s + c$; otherwise we schedule the tail job at slot s and then reschedule its predecessor.

All of the scheduling decisions are made in procedure SCHED, which can be invoked with the parameter $r = 0$ or $r = 1$. When we try to schedule a job S_i which has not been scheduled before, SCHED is invoked with $r = 0$. When SCHED is invoked with $r = 1$, it reschedules a job S_i (which will be moved to a new slot) by inspecting the slot t. The feasibility of the schedules produced by the algorithm is shown next.

LEMMA 3.3. *If SMD terminates successfully without reporting "unschedulable," the schedule generated by the algorithm is a feasible schedule for the job set.*

Proof. From the discussion in §3.1, it is easy to see that each successful iteration of Step 2 in SMD will produce a well-formed partial schedule. If the algorithm terminates successfully without reporting "unschedulable" then all jobs must have been scheduled with all constraints satisfied. Hence, the final schedule produced is feasible. □

THEOREM 3.4. *Algorithm SMD has a time complexity of* $O(n^2)$.

Proof. Step 1 (sorting) can be done in $O(n \log n)$. In Step 2 the algorithm will recursively call SCHED with the second parameter setting to $t - 1$ if and only if time slot t has already been assigned to some other job. Since the total number of jobs is n, the total number of recursive calls is at most $O(n)$. This means that Step 2 can be done in $O(n^2)$ with carefully designed data structures. Therefore, the time complexity of Algorithm SMD is $O(n^2)$. □

3.3. The schedulability condition for MUJSD systems. In Lemma 3.3, we show that if SMD produces a schedule for an MUJSD system, then the schedule satisfies all of the precedence, deadline, and distance constraints for the job set of the MUJSD system. In this section, we show that if an MUJSD system is schedulable, then SMD will find a feasible schedule for it. In other words, SMD reports a failure only if the job set is unschedulable.

To find a sufficient scheduling condition for MUJSD systems, we need a more dynamic notion that can be used to discuss the partial schedules produced by SMD. For each partial schedule produced by SMD, we define an *effective deadline* for each job in the system as follows.

DEFINITION. *Given a partial schedule f, if job J_{ix} has been scheduled, its effective deadline is defined as $f(J_{ix}) + 1$. If J_{ix} is a head job and has not yet been scheduled, its effective deadline is defined as its deadline d_i. If J_{ix} is a tail job and has not yet been scheduled, its effective deadline is the effective deadline of its predecessor $J_{i,x-1}$ plus c.*

Before the final schedule is produced, the effective deadline of a job may be changed whenever SMD (re)schedules the job itself or one of its predecessors. In SMD, when SCHED tries to move a job from slot t to slot $t - 1$, the job to be scheduled at slot $t - 1$ thus has a new deadline constraint of $t - 1$. In other words, the effective deadline of the job is changed to $t - 1$. Moreover, the effective deadlines of its unscheduled successors are affected accordingly.

One of the properties of SMD is that once a slot is assigned to a job, the slot will never become empty again. Moreover, during the execution of SMD, if job S_i is scheduled at slot t

and its immediate successor S_j is also scheduled in the partial schedule, then slot $t + c$ must be nonempty. This can be easily seen from the algorithm.

For the rest of the discussion, to distinguish the different states when running SMD, when we refer to a partial schedule f, we actually include the current effective deadlines of all the jobs (scheduled or not) in the system. In the following defintion, we define $u(f, d)$ as the number of unscheduled jobs with effective deadlines less than or equal to d in the partial schedule f, and $e(f, d)$ as the number of empty slots before time d in f.

DEFINITION. *A partial schedule f is said to be* extensible *if $e(f, d) \geq u(f, d)$ for all $d > 0$.*

It is obvious that when two or more jobs are competing for a slot (i.e., these jobs have the same effective deadline), only one of them can be scheduled at the slot; the others must be scheduled at some earlier slots.

LEMMA 3.5. *In a partial schedule f, suppose S_i is an unscheduled job with an effective deadline d and $e(f, d) - u(f, d) \geq 0$. If there is another job with the same effective deadline d, we can derive another schedule g from f by changing the effective deadline of S_i from d to $d - 1$ and maintain the condition that $e(g, d) - u(g, d) \geq 0$. Furthermore, it is also true that $e(g, d - 1) - u(g, d - 1) \geq 0$.*

Proof. Note that S_i is unscheduled in both f and g, but it has different effective deadlines in them. It is obvious that $e(g, d) = e(f, d)$ and $u(g, d) = u(f, d)$. Therefore, we have $e(g, d) - u(g, d) = e(f, d) - u(f, d) \geq 0$.

If slot d is nonempty in f (and thus in g) then $e(f, d) = e(f, d - 1) = e(g, d - 1)$ and $u(g, d - 1) \leq u(f, d)$. Therefore, $e(g, d - 1) - u(g, d - 1) \geq e(f, d) - u(f, d) \geq 0$. If slot d is empty in f then there must be another unscheduled job with effective deadline d. We have $e(g, d - 1) \geq e(f, d) - 1$ and $u(g, d - 1) \leq u(f, d) - 1$. Again, $e(g, d - 1) - u(g, d - 1) \geq e(f, d) - u(f, d) \geq 0$. □

We have shown the feasibility of the partial schedule generated after each iteration in Step 2 of SMD. Also note that if the partial schedule, generated after S_{i-1} is scheduled, is extensible, by Lemma 3.2 and the definition of extensibility, we know that S_i can always be feasibly scheduled. Therefore, if we can show that the partial schedule, generated after S_i is scheduled, is still extensible, then we can prove the correctness of algorithm SMD.

LEMMA 3.6. *In Step 2 of SMD, if the current partial schedule before an iteration of the loop is extensible, then the partial schedule produced after the iteration is still extensible.*

Proof. Suppose that after S_{i-1} is scheduled, the partial schedule f produced by SMD is extensible. We now prove that the partial schedule, produced after S_i is scheduled, is also extensible. To prove this it suffices for us to show that every operation in SMD that changes the effective deadline of a job from t to $t - 1$ will not affect the extensibility of the partial schedule. There are two cases to be considered.

Case 1. This case involves scheduling the current job S_i ($r = 0$ in SCHED).

Suppose we are trying to schedule S_i at time slot t. If slot t is empty we will schedule S_i at slot t, which does not affect the extensibility of the partial schedule. If slot t is occupied by S_k and S_k is the immediate predecessor of S_i, we will schedule S_i at slot t and try to reschedule S_k at slot $t - 1$ by calling SCHED with $r = 1$. By Lemma 3.5, scheduling S_i at slot t and rescheduling S_k by changing the effective deadline of S_k to $t - 1$ will not affect the extensibility of the partial schedule. If slot t is occupied by S_k and S_k is not the immediate predecessor of S_i, we will try to schedule S_i at slot $t - 1$ by calling SCHED with $r = 0$. This changes the effective deadline of S_i to $t - 1$ and the effective deadlines of all its successors, if any, to one unit time smaller. We next prove that changing the effective deadlines of S_i and its successors will not affect the extensibility of the partial schedule.

Since S_i is the job to be scheduled and S_k is a job that has already been scheduled, we must have $k < i$, that is, the number of successors of S_k is no less than that of S_i. If none of S_k's successors has been scheduled, then for each of S_i's successors there is a corresponding successor job of S_k that has the same effective deadline. On the other hand, if the immediate successor of S_k has been scheduled, then slot $t + c$ must be occupied by some job S_z with $z < i$. Repeating the above argument, we can see that for job S_i and each of its successor jobs, either there is another unscheduled job with the same effective deadline or the time slot for the effective deadline is nonempty. With these conditions, by Lemma 3.5, changing the effective deadlines of S_i and all its successor jobs to one unit time smaller will not affect the extensibility of the partial schedule.

Case 2. This case involves rescheduling a job S_j ($r = 1$ in SCHED).

Suppose we are trying to schedule S_j at time slot t. If slot t is empty or occupied by the immediate predecessor of S_j, the situation is similar to that in Case 1. If slot t is occupied by S_k and S_k is not the immediate predecessor of S_j, then we will schedule either S_j or S_k at slot t and try to reschedule the other one at slot $t - 1$. In the following proof, we use S_p for the job to be rescheduled and S_q for the job now scheduled at t. Scheduling S_q at slot t does not change its effective deadline so that the extensibility of the partial schedule is not affected. We only need to show that rescheduling S_p by changing its effective deadline to $t - 1$ will also not affect the extensibility of the partial schedule.

If S_p's immediate successor has been scheduled, then changing S_p's effective deadline to $t - 1$ will not change the effective deadline of any other unscheduled job, and by Lemma 3.5, this does not affect the extensibility of the partial schedule, since slot t is nonempty (now occupied by S_q). On the other hand, if none of S_p's successors has been scheduled, then S_p, S_i, and S_p's immediate successor have nonincreasing numbers of successors. Since S_p was to be rescheduled because of the operation of rescheduling some other job, we must have $t + c < s$, where s is the first slot in which we tried to schedule S_i. This means slot $t + c$ must be nonempty at that moment. Suppose S_y is currently scheduled at slot $t + c$. Since the number of successors of S_y is no less than the number of successors of S_i, it is no less than the number of successors of S_p minus one. Using an argument similar to that in Case 1, we can show that, for job S_p and each of its successor jobs, either there is an unscheduled job with the same effective deadline or the effective deadline slot is nonempty. Then, by Lemma 3.5, changing the effective deadlines of S_p and all its successor jobs to one unit time smaller will not affect the extensibility of the partial schedule. □

LEMMA 3.7. *For any MUJSD problem with n jobs, if* $h(d) \leq d$ *for all* d, $1 \leq d < n$, *then SMD will find a feasible schedule for it.*

Proof. Initially we have $u(f, d) = h(d) \leq d = e(f, d)$ for all $d > 0$, where f is the empty schedule before Step 2. Thus the empty schedule f is an extensible partial schedule. Moreover, S_1 can be feasibly scheduled into f, and by Lemma 3.6, the partial schedule produced is still extensible. Similarly, we can show that in each iteration of Step 2, S_i can always be feasibly scheduled and the partial schedule produced is still extensible. Therefore, all jobs can be feasibly scheduled, i.e., Algorithm SMD will finally find a feasible schedule for this MUJSD problem. □

4. A polynomial-time algorithm for the MUJSD problem. In this section, we present the PMD algorithm which will generate a feasible schedule for a schedulable MUJSD system with m job chains and distance constraint c in $O(m^2c^2)$ time. We first introduce the data structures used in the algorithm. The algorithm is then presented in detail, including an example on job rescheduling. We also prove the properties and the time complexity of the algorithm.

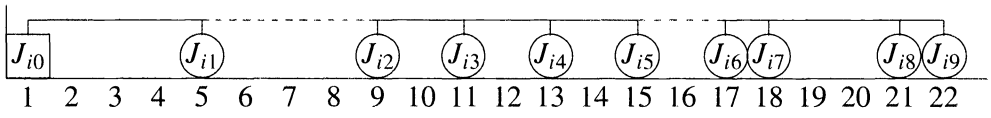


FIG. 4.1. *The schedule corresponds to the S-list* [(9; 22; 1, 3), (5; 15; 2, 2), (1; 5; 4)].

4.1. Data structures. To have a polynomial-time algorithm, the output (i.e., the schedule) from the algorithm cannot specify the start time of each job J_{ij}. Instead, we must use a data structure to describe the schedule for each job chain B_i. We use a list, called *S-list*, to describe the schedule for each chain in an MUJSD system. Every element in the S-list is of the form $(j; s; c_1, c_2, \ldots, c_l)$, where $\sum_{i=1}^{l} c_i = c$. The c_i's define the distance patterns between jobs. An S-list specifies the job positions backwards. Suppose S-list S_i (the S-list for the job chain B_i) is $[\ \ldots, (j; s; c_1, c_2, \ldots, c_l), (j'; s'; c'_1, c'_2, \ldots, c'_k), \ldots\ ]$; this means that J_{ij} is scheduled at slot s, $J_{i,j-1}$ is scheduled at c_1 time units before J_{ij} (i.e., slot $s - c_1$), $J_{i,j-2}$ is scheduled at c_2 time units before $J_{i,j-1}$ (i.e., slot $s - c_1 - c_2$), ..., and $J_{i,j-l}$ is scheduled at c_l time units before $J_{i,j-l+1}$. The pattern repeats for those preceding jobs up to job $J_{i,j'+1}$, since j' is defined in the next element. Job $J_{i,j'}$ starts a new distance pattern, starting at slot s'.

Figure 4.1 shows the schedule corresponding to the S-list [(9; 22; 1, 3), (5; 15; 2, 2), (1; 5; 4)] for job chain B_i (note that distance constraint $c = 4$). The first element in the S-list is (9; 22; 1, 3), which means that J_{i9} is scheduled at slot 22, J_{i8} is scheduled at slot 21 ($= 22 - 1$), J_{i7} is scheduled at slot 18 ($= 21 - 3$), and J_{i6} is scheduled at slot 17 ($= 18 - 1$). Since the second element in the S-list is (5; 15; 2, 2), the scheduled position of J_{i5} is defined by the second element, not the first element. That is, J_{i5} is scheduled at slot 15. Similarly, J_{i4} is scheduled at slot 13 ($= 15 - 2$), J_{i3} is scheduled at slot 11 ($= 13 - 2$), and J_{i2} is scheduled at slot 9 ($= 11 - 2$). The last element (1; 5; 4) specifies that J_{i1} is scheduled at slot 5 and J_{i0} is scheduled at slot 1 ($= 5 - 4$). It is obvious that, given a valid S-list, there is exactly one schedule corresponding to it.

For each job chain B_i, we define two deadlines as follows.

DEFINITION. *The* head deadline *of the chain B_i is the deadline d_i of the head job H_i. The* tail deadline *of B_i is the virtual deadline $d_{ik_i} = d_i + k_i \cdot c$ of the last tail job J_{ik_i}.*

Since the PMD algorithm must work on a sequence of time slots, not individual slots, we partition time slots into groups. Let $D_{\max} = \max_{1 \le i \le m}\{d_{ik_i}\}$ be the latest tail deadline of all job chains. It is obvious that no job can be scheduled after $D_{\max}$. We partition all of the time slots before $D_{\max}$ (including slots before time 0) into c different groups, which we call *E-groups* E_i, $1 \le i \le c$. Let $D_i = D_{\max} - i + 1$ for $1 \le i \le c$. E_i consists of time slots D_i, $D_i - c$, $D_i - 2c$, ... When we schedule the jobs in the PMD algorithm, the slots in an E-group will be filled from the tail. We call a sequence of empty slots $s, s - c, s - 2c, \ldots$ in an E-group a *v-chain* with the *tail position* s. Initially, each E-group E_i is a v-chain with tail position D_i. We call it v-chain V_i. In the algorithm, these v-chains will be filled up from the tail so that they become shorter and shorter.

4.2. The PMD algorithm. We now present the PMD algorithm. In Step 1, we sort the job chains and reindex them so that the chain with a larger tail deadline has a larger index (ties are broken arbitrarily). Also, we create a pseudochain 0 which has only one job with a deadline 0 (note that the head deadlines of all the other job chains are larger than 0). This pseudochain serves as a marker to trigger the final cleanup process which will move jobs scheduled before time 0 to empty slots after time 0. Step 2 sets the initial tail positions of the v-chains and initializes the counter p which points to the current job chain being scheduled.

Step 3 is the main step of the algorithm where we try to fit each job chain into some v-chain. We process the job chains one at a time starting from the chain with the latest tail deadline. We try to schedule job chain B_p into the v-chain V_y, which has the latest tail position D_y. If the tail deadline d_{pk_p} of B_p is larger than or equal to D_y, we can schedule the whole job chain in a single assignment. If $p > 0$, we create an S-list $[(k_p; D_y; c)]$ for B_p, and reset the tail position of V_y to $D_y - (k_p + 1)c$. If $p = 0$, we have already scheduled chains 1 to m and all the tail positions of the v-chains are less than or equal to 0. Therefore, we can go to Step 4 to check if the schedule that is found is feasible.

If the tail deadline d_{pk_p} of B_p is less than D_y, slot D_y cannot be used by any job in B_1 to B_p. However, there may be some previously scheduled jobs that can be rescheduled at D_y. Therefore, we search for any scheduled job chain B_i which has at least one job scheduled before D_y with a virtual deadline $\geq D_y$. We then reschedule some of the jobs in B_i. The rescheduling procedure will be discussed in detail in the next section. If we cannot find such a job chain, then we reset the tail position of V_y to $D_y - \lceil (D_y - d_{pk_p})/c \rceil c$. Step 3 is then repeated to schedule job chain B_p.

Step 4 checks if the schedule that is found is feasible. We only need to check if all head jobs are scheduled after time 0. If any of the head jobs is scheduled before time 0 then the schedule is infeasible. Given an S-list S_i, it is easy to find the scheduled slot of the head job J_{i0} in at most $O(c)$ time. Therefore, Step 4 can be done in time $O(mc)$.

ALGORITHM PMD

Step 1. Sort and reindex the job chains in nondecreasing tail deadline order
(i.e., $d_{ik_i} \leq d_{i+1,k_{i+1}}$, for $1 \leq i < m$);
Create a pseudochain B_0 with $d_0 = 0$ and $k_0 = 0$;

Step 2. Set $D_1 = d_{mk_m}$;
For $i = 2$ to c do { set $D_i = D_{i-1} - 1$; }
Set $p = m$;

Step 3. Let y be the index of the v-chain with $D_y = \max_{1 \leq i \leq c} D_i$;
If $d_{pk_p} \geq D_y$
then { /* schedule chain p */
If $p = 0$ goto Step 4;
initialize S-list S_p to be $[(k_p; D_y; c)]$;
reset $D_y = D_y - (k_p + 1)c$;
set $p = p - 1$; }
else { /* reschedule chains */
Among scheduled chains $p + 1$ to m find a chain B_i,
and locate the job J_{ij}, $j \geq 0$, where
(S3.1) J_{ij} is the latest job in B_i scheduled before D_y, and
(S3.2) $d_{ij} \geq D_y$.
If B_i and job J_{ij} exist
then reschedule jobs $J_{i0}, J_{i1}, \ldots, J_{ij}$; /* as in §4.3 */
else reset $D_y = D_y - \lceil (D_y - d_{pk_p})/c \rceil c$;
}
Repeat Step 3.

Step 4. If there is any head job scheduled before time 0
then output "unschedulable";
else output the m S-lists.

4.3. Rescheduling job chains. In Step 3 of PMD, after job J_{ij} is identified, we want to reschedule it and all its predecessor jobs to some new slots. To have a polynomial-time

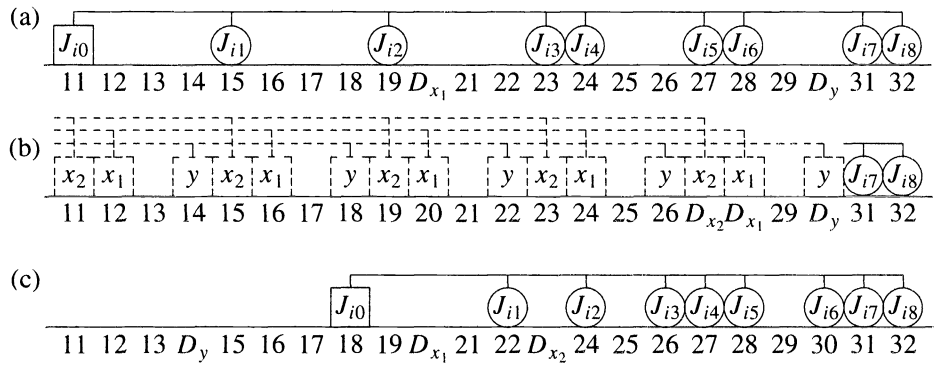


FIG. 4.2. *An example of the rescheduling procedure of Algorithm PMD.*

algorithm we must reschedule the jobs altogether by computing the resulting S-list directly, not individually. We first remove jobs $J_{i0}, J_{i1}, \ldots, J_{ij}$ from the schedule. This is done by returning all their slots to the corresponding v-chains. We then try to fill the jobs backwards into the available slots as late as possible (starting from D_y). However, we still need to satisfy the deadline requirement of J_{i0}. Therefore we may have more than one distance pattern for the rescheduled jobs. Let us first use an example to illustrate the rescheduling procedure (Fig. 4.2).

Example. Suppose $d_{pk_p} < D_y = 30$ and $c = 4$. V_y has a tail position $D_y = 30$ and consists of slots 30, 26, 22, ... Also suppose S_i is [(8; 32; 1, 3), (2; 19; 4)] and $d_{i0} = 18$ (Fig. 4.2(a)). J_{i6} is currently scheduled at slot 28 and has a virtual deadline $d_{i6} = 18+6\cdot 4 = 42$. Therefore, J_{i6} can be rescheduled at slot D_y. (Note that J_{i7} is scheduled at slot $31 > D_y$.) Let the v-chain with tail position 20 be v-chain V_{x_1}, and the v-chain with tail position 7 be v-chain V_{x_2}, i.e., $D_{x_1} = 20$ and $D_{x_2} = 7$. Note that slots in V_{x_1} and V_{x_2} are in the same E-groups as those slots currently occupied by jobs $J_{i0}, J_{i1}, \ldots, J_{i6}$. If we "unschedule" jobs $J_{i0}, J_{i1}, \ldots, J_{i6}$, the tail positions D_{x_1} and D_{x_2} will become 28 and 27, respectively (Fig. 4.2(b)).

We will reschedule jobs $J_{i0}, J_{i1}, \ldots, J_{i6}$ at only those slots in V_y, V_{x_1}, and V_{x_2} as late as possible. However, they still must satisfy the deadline and distance constraints. In other words, J_{i6} is rescheduled at slot 30, J_{i5} is rescheduled at slot 28, J_{i4} is rescheduled at slot 27, J_{i3} is rescheduled at slot 26, and J_{i2} is rescheduled at slot 24. Ideally, we would like to reschedule J_{i1} at slot 23. But since the deadline of J_{i0} is 18, J_{i1} cannot be scheduled at a slot later than $22 = 18 + c$. Therefore, the best we can do is schedule J_{i0} at slot 18 and J_{i1} at slot 22. The final schedule for these jobs is shown in Fig. 4.2(c). The new S-list for the ith chain becomes [(8; 32; 1, 3), (6; 30; 2, 1, 1), (1; 22; 4)], and the tail positions D_y, D_{x_1}, and D_{x_2} of V_y, V_{x_1}, and V_{x_2} now become 14, 20, and 23, respectively.

Before we present the rescheduling procedure, we need to define some notation. Let J_{ij} be the job that satisfies conditions (S3.1) and (S3.2) in Step 3 of PMD. Suppose the current S_i has a form $[(j_1; s_1; c_{11}, c_{12}, \ldots, c_{1l_1}), \ldots, (j_q; s_q; c_{q1}, c_{q2}, \ldots, c_{ql_q})]$, and J_{ij}'s scheduled position is defined in the uth element $(j_u; s_u; c_{u1}, c_{u2}, \ldots, c_{ul_u})$. In Lemma 4.1 we will show that

(1) $u = q$ or $q - 1$;
(2) if there is more than one element in S_i, J_{ij} is not scheduled at s_{q-1};
(3) if there is only one element in S_i, J_{ij} is not scheduled at s_q;
(4) if $j \geq j_q$ then the last element in S_i has a simple distance pattern $(j_q; s_q; c)$.

Suppose there are L distance patterns in the uth element of S_i, i.e., $L = l_u$. Let us rename the c_{uv}'s as C_v's, i.e., $C_v = c_{uv}$ for $v = 1, 2, \ldots, L$. Also, we define $C_0 = C_L$. Suppose job J_{ij}

is currently scheduled at slot K. We will define some special E-groups for our discussion as follows:

(1) If $j \neq j_q$, and we suppose that the current distance pattern between J_{ij} and $J_{i,j+1}$ is C_r, $0 \leq r < L$ (i.e., $K = s_u - Q \cdot c - \sum_{v=1}^{r} C_v$ for some integer Q), we will call the E-groups that slots K, $K - C_{r+1}$, $K - C_{r+1} - C_{r+2}$, ..., $K - \sum_{v=r+1}^{L} C_v$, $K - \sum_{v=r+1}^{L} C_v - C_1$, ..., and $K - \sum_{v=r+1}^{L} C_v - \sum_{v=1}^{r-1} C_v$, belong to as E_{x_1}, E_{x_2}, ..., and E_{x_L}, respectively.

(2) If $j = j_q$, we know that $L = 1$ and $C_L = c$ from Lemma 4.1. We will define $r = 1$ (hence, $C_r = C_L = c$) and call the E-group to which K belongs E_{x_1}.

Finally, we will rename y as x_0 so that $D_{x_0} = D_y$.

The rescheduling procedure has two parts. In the first part, we unschedule jobs J_{i0}, J_{i1}, ..., and J_{ij}. To do this, there are three cases. If $j > j_q$ (i.e., $u = q - 1$) we delete the last element in S_i, i.e., element $(j_q; s_q; c_{q1})$. If $j = j_q$ (so $u = q$) we again delete the last element in S_i, since we are going to reschedule all the jobs defined in this element. Finally, if $j < j_q$ (again $u = q$) we don't need to delete any element. In Lemma 4.1, we will prove that jobs $J_{i0}, J_{i1}, \ldots, J_{ij}$ are currently scheduled at the slots that belong to the L E-groups $E_{x_1}, E_{x_2}, \ldots, E_{x_L}$ (condition (c)), and if we unschedule these jobs, their slots will merge with the existing v-chains in the L E-groups (condition (d)). Therefore, we need to reset D_{x_1}, D_{x_2}, ..., D_{x_L} to K, $K - C_{r+1}$, $K - C_{r+1} - C_{r+2}$, ..., $K - \sum_{v=r+1}^{L} C_v$, $K - \sum_{v=r+1}^{L} C_v - C_1$, ..., and $K - \sum_{v=r+1}^{L} C_v - \sum_{v=1}^{r-1} C_v$, respectively. These L v-chains and the v-chain V_{x_0} with tail position $D_{x_0} = D_y$ are the only v-chains that will be used in the second part of the rescheduling. That is, we try to reschedule jobs $J_{i0}, J_{i1}, \ldots, J_{ij}$ at only these $L + 1$ v-chains and the latest possible slots.

In part two of the rescheduling, among the $L + 1$ v-chains, we find the latest slot H before the head deadline d_i of B_i (i.e., $H = \max\{t \mid t \leq d_i \text{ and } D_{x_v} - t = 0 \text{ (modulo } c), 0 \leq v \leq L\}$). Suppose H is in v-chain V_{x_α}, i.e., $D_{x_\alpha} - H = \delta \cdot c$ for some integer δ. We know that there are $\delta(L + 1) + (\alpha + 1)$ empty slots available between slots H and D_{x_0} for rescheduling the $j + 1$ jobs. We want to reschedule the jobs at the latest possible slots, yet J_{i0} must be scheduled at or before H. There are three possibilities:

(1) If $\alpha = 0$ and $D_{x_0} - H = j \cdot c$, J_{ij} should be rescheduled at slot $D_{x_0} = D_y$. Each J_{iv}, $0 \leq v < j$, should be scheduled at c time units before $J_{i,v+1}$. In this way, J_{i0} will be rescheduled at slot $D_{x_0} - j \cdot c = H$, which is the latest slot in which J_{i0} can be scheduled. In terms of the data structure, we append the element $(j; D_{x_0}; c)$ to the end of S_i and reset $D_{x_0} = H - c$. Since $D_{x_1}, D_{x_2}, \ldots, D_{x_L}$ have been reset to appropriate values, they don't need to be changed.

(2) If $j + 1 \geq \delta(L + 1) + (\alpha + 1)$, then the total number of jobs to be rescheduled is larger than or equal to the total number of empty slots available from slot H to slot D_{x_0}. Therefore, J_{i0} will automatically be scheduled at slot H or earlier. As for the data structure, we append the element $(j; D_{x_0}; D_{x_0} - K, C_{r+1}, C_{r+2}, \ldots, C_L, C_1, C_2, \ldots, C_{r-1}, C_r - D_{x_0} + K)$ to the end of S_i. Also, assume that $(j + 1) = \lambda(L + 1) + \gamma$ for some integer λ and $0 \leq \gamma < (L + 1)$. We reset D_{x_v} to $D_{x_v} - (\lambda + \rho)c$ for $v = 0, 1, \ldots, L$, where $\rho = 1$ if $\gamma > v$ and $\rho = 0$ otherwise.

(3) If $j + 1 < \delta(L + 1) + (\alpha + 1)$, then the number of jobs to be rescheduled is less than the number of empty slots available. In this case, we still need to reschedule job J_{i0} at slot H or earlier because of the deadline constraint. From earlier discussions, we know that $D_{x_0} - H = (D_{x_\alpha} - H) + (D_{x_0} - D_{x_\alpha}) = \delta \cdot c + (D_{x_0} - D_{x_\alpha})$ (note that $0 \leq D_{x_0} - D_{x_\alpha} < c$). Now, if $j - \delta < \alpha$, then if we try to reschedule jobs $J_{ij}, J_{i,j-1}, \ldots, J_{i0}$ one by one at the available empty slots from D_{x_0} to H, we will schedule job $J_{i\delta}$ at a slot later than D_{x_α}. But since job $J_{i\delta}$ cannot be scheduled at

a slot later than D_{x_α}, we schedule it at slot D_{x_α}. Therefore, the schedule pattern changes from job $J_{i\delta}$. If $j - \delta < \alpha$, we define $j' = \delta$ and $s' = D_{x_\alpha}$, or else define $j' = (j - \alpha) - \lambda(L + 1) - (\gamma + 1)$ and $s' = D_{x_\alpha} - \lceil (j - \alpha - \delta)/L \rceil c$, where $j - \alpha - \delta = \lambda \cdot L + \gamma$ for some integers λ and $0 \le \gamma < L$. We thus append elements $(j; D_{x_0}; D_{x_0} - K, C_{r+1}, C_{r+2}, \ldots, C_L, C_1, C_2, \ldots, C_{r-1}, C_r - D_{x_0} + K)$ and $(j'; s'; c)$ to the end of S_i. We must also update $D_{x_0}, D_{x_1}, \ldots, D_{x_L}$ as follows. If $j - \delta < \alpha$, D_{x_v} is reset to $D_{x_v} - c$ for $0 \le v < j - \delta$ and D_{x_α} is reset to $H - c$ (D_v, for $j - \delta \le v \le L$ and $v \ne \alpha$, remain unchanged since they have been reset to the appropriate values already). If $j - \delta \ge \alpha$, D_{x_v} for $0 \le v < \alpha$ is reset to $D_{x_v} - (\lambda + 1 + \lceil (\gamma - L + \alpha - v)/L \rceil)c$, D_{x_α} is reset to $H - c$, and D_{x_v} for $\alpha < v \le L$ is reset to $D_{x_v} - (\lambda + \lceil (\gamma - (v - \alpha - 1))/L \rceil)c$.

After we have rescheduled the jobs in B_i we repeat Step 3 to schedule the current B_p. It is easy to see that after the rescheduling the tail positions of some of the v-chains may increase, but $\max_{1 \le v \le c} D_v$ should decrease, since the tail positions of the v-chains involved in the rescheduling procedure will get smaller values than the original D_y.

4.4. Properties of Algorithm PMD. In this section, we prove the correctness and the time complexity of Algorithm PMD.

LEMMA 4.1. *During the execution of Algorithm PMD, suppose job chain B_a has been scheduled and suppose the S-list S_a is $[(j_1; s_1; c_{11}, c_{12}, \ldots, c_{1l_1}), \ldots, (j_q; s_q; c_{q1}, c_{q2}, \ldots, c_{ql_q})]$. Also, suppose V_y is the v-chain with the largest tail position D_y among all v-chains. Let $J_{a0}, J_{a1}, \ldots, J_{ab}$, $b \ge 0$, be the jobs in B_a currently scheduled before D_y. The following conditions must be true:*

(a) *If $q > 1$ then $b < j_{q-1}$ and if $q = 1$ then $b < j_q$.*

(b) *If $b \ge j_q$ then $l_q = 1$ (hence $c_{q1} = c$).*

Suppose J_{ab} is scheduled at slot K which is defined in the uth element of S_a, i.e., $u = q - 1$ if $j_q < b < j_{q-1}$ and $u = q$ if $b \le j_q$. Let $L = l_u$ be the number of distance patterns in the element, and let us rename $C_v = c_{uv}$ for $v = 1, 2, \ldots, L$ and $C_0 = C_L = c_{uL}$. If $b \ne j_q$, we can find two integers Q and r, $0 \le r < L$, such that the current distance between J_{ab} and $J_{a,b+1}$ is C_r, and $K = s_u - Q \cdot c - \sum_{v=1}^{r} C_v$. Let slots K, $K - C_{r+1}$, $K - C_{r+1} - C_{r+2}, \ldots, K - \sum_{v=r+1}^{L} C_v$, $K - \sum_{v=r+1}^{L} C_v - C_1, \ldots$, and $K - \sum_{v=r+1}^{L} C_v - \sum_{v=1}^{r-1} C_v$ belong to the E-groups $E_{x_1}, E_{x_2}, \ldots$, and E_{x_L}, respectively. If $b = j_q$, from (b) *we know that $L = 1$ and $C_L = C_0 = c$. In this case, let E_{x_1} be the E-group to which K belongs and let $r = 1$ (hence, $C_r = C_L = c$). The following conditions must also be true:*

(c) *If $b > j_q$ then slot $s_q \in E_{x_v}$ for some $1 \le v \le L$.*

(d) *For each v, $1 \le v \le L$, let $J_{ab'}$ be the first job in chain a scheduled at slot s' and $s' \in E_{x_v}$. If there exists such a b' then $D_{x_v} = s' - c$; otherwise, $s_u - c < D_{x_v} < s_u$.*

Proof. It is obvious that the initial empty schedule satisfies conditions (a)–(d). Suppose conditions (a)–(d) are true before an iteration of Step 3. We now show that after the iteration they still hold. If $d_{pk_p} \ge D_y$, we will schedule B_p at V_y by setting S_p to $[(k_p; D_y; c)]$ and decrease the value of D_y. It is easy to see that conditions (a)–(d) still hold, since the value of $\max_{1 \le v \le c} D_v$ is smaller than the original D_y. If $d_{pk_p} < D_y$ and no job chain satisfies conditions (S3.1) and (S3.2) in Step 3, we will only decrease the value of D_y, hence conditions (a)–(d) still hold. If we invoke the rescheduling procedure, the only job chain that will be affected is B_i. Because B_i satisfies conditions (a)–(d), the rescheduling procedure is valid as discussed in the previous section. Moreover, although the tail positions of some v-chains may increase, the value $\max_{1 \le v \le c} D_v$ will only decrease. Therefore, conditions (a)–(d) still hold for all of the scheduled chains except B_i. However, after the rescheduling procedure, J_{ij} will be scheduled at the original D_y and it is not hard to see from the rescheduling procedure that all four conditions still hold for B_i after the rescheduling. □

LEMMA 4.2. *In Step* 3 *of Algorithm PMD, if* $d_{pk_p} < D_y$ *and none of the scheduled chains satisfies conditions* (S3.1) *and* (S3.2), *then none of the jobs currently scheduled before slot* $D_y - v \cdot c$ *has a virtual deadline larger than or equal to* $D_y - v \cdot c$ *for each* $v = 1, 2, \ldots$.

Proof. From the algorithm we know that if a scheduled chain B_i has a job scheduled before slot D_y, then it must have at least one job scheduled after slot D_y. Let $J_{i,j+1}$ be the first job in chain i that is scheduled after slot D_y. Suppose there is a job $J_{ij'}$ with virtual deadline larger than or equal to $D_y - v \cdot c$ and currently scheduled before slot $D_y - v \cdot c$ for some $v \geq 1$. We must have $(j+1) - j' \geq v + 1$, i.e., $j - j' \geq v$. This means that J_{ij}, which is scheduled before slot D_y, has a virtual deadline larger than or equal to D_y. Hence, B_i satisfies conditions (S3.1) and (S3.2), which contradicts the assumption. □

THEOREM 4.3. *Algorithm PMD will find a feasible schedule if the system is schedulable and will output "unschedulable" if the system is unschedulable.*

Proof. From the algorithm, we know that the final schedule generated by Algorithm PMD has no two jobs occupying the same slot. Moreover, the schedule satisfies the deadline, distance, and precedence constraints. Now, the only thing we need to prove is that if the system is schedulable, none of the jobs will be scheduled before time 0. If there is no empty slot from slot 1 to slot D in the final schedule, where $D = \max_{1 \leq i \leq m} d_{ik_i}$, then a job scheduled before time 0 implies that the total number of jobs in the system is larger than D, which in turn implies that the system is unschedulable. Suppose there are empty slots before time D in the generated schedule. Let t be the first empty slot. From Lemma 4.2, it is easy to see that none of the jobs scheduled before slot t has a virtual deadline larger than or equal to t. If there is a job scheduled before time 0, then the total number of jobs with virtual deadlines less than t is larger than $t - 1$, hence the system is unschedulable. □

LEMMA 4.4. *In Step* 3 *of Algorithm PMD, finding a chain that satisfies conditions* (S3.1) *and* (S3.2) *or making sure that none of the chains satisfies the two conditions can be done in time at most* $O(mc)$.

Proof. By Lemma 4.1, any chain i that satisfies conditions (S3.1) and (S3.2) must have $j < j_{q-1}$. That is, we only need to check for each of the chains $p+1$ to m, the last two elements of its associated S-list. To locate the job J_{ij} in chain i that satisfies the two conditions needs time at most $O(c)$, since the c-pattern of each element in the S-list has at most c components. Therefore, the total time needed is at most $O(mc)$. □

THEOREM 4.5. *The time complexity of Algorithm PMD is at most* $O(m^2c^2)$.

Proof. The time complexity of the algorithm is dominated by Step 3. Finding D_y needs time $O(c)$ and we will show that Step 3 will be repeated at most $O(mc)$ times, therefore, the total time spent on finding $\max_{1 \leq v \leq c} D_v$ is $O(mc^2)$. Initializing S-list S_p can be done in $O(1)$ time and there are total m S-lists, therefore, the total time spent on the "then" part of the test "$d_{pk_p} \geq D_y$" is $O(m)$. From Lemma 4.4, finding a chain B_i that satisfies (S3.1) and (S3.2) or making sure that no such chain exists needs time at most $O(mc)$. Rescheduling a chain needs time at most $O(c)$. Since each job chain will be rescheduled with respect to each v-chain at most once, the rescheduling procedure will be executed at most $O(mc)$ times and Step 3 will be repeated at most $O(mc)$ times. Therefore, the total rescheduling procedure takes time at most $O(m^2c^2)$. Since this complexity dominates the whole algorithm, the complexity of the algorithm is at most $O(m^2c^2)$. □

Given an MUJSD system, if we want to know the schedulability of the system, we can use the PMD algorithm, which takes time $O(m^2c^2)$ to find a schedule for it. If there exists a feasible schedule for the system, then it is schedulable; otherwise, it is unschedulable. In fact, it can be shown that the schedulability test for an MUJSD system can be done in time $O(m^2c)$ [7].

5. Some extensions to the MUJSD problem. In the MUJSD problem, we assume all deadlines and distance constraints are integers. If we allow deadlines and distance constraints to be fractional numbers, both SMD and PMD can still be used. If we change all deadlines d_i, $1 \le i \le n$, and the distance constraint c in an MUJSD system to $\lfloor d_i \rfloor$ and $\lfloor c \rfloor$, respectively, the modified system is schedulable if and only if the original system is schedulable.

The MUJSD problem also assumes that all distance constraints are the same. In the following discussion, we investigate several related problems which extend MUJSD in terms of the distance constraint values, job set structure, and the number of processors. We show that these problems are NP-complete.

5.1. The general MUJSD problem. One extension of the MUJSD problem is to allow different jobs to have different distance constraints instead of having only an identical distance constraint c for all jobs. We call this extension the *general* MUJSD problem. However, we can show that the general MUJSD problem is NP-complete.

THEOREM 5.1. *The extension of the MUJSD problem, where different chains may have different distance constraints, is NP-complete in the strong sense.*

Proof. It is easy to see that this problem is in NP. To complete the proof, we reduce the 3-PARTITION problem [5], which has been shown to be NP-complete in the strong sense, to the problem. The 3-PARTITION problem can be stated as follows: Given a positive integer b and a multiset $\mathbf{A} = \{a_1, a_2, \ldots, a_{3l}\}$ of $3l$ positive integers such that $\sum_{i=1}^{3l} a_i = l \cdot b$ and $b/4 < a_i < b/2$ for each $1 \le i \le 3l$, can $\mathbf{A}$ be partitioned into l disjoint sets $S_1, S_2, \ldots, S_l$ such that, for $1 \le j \le l$, $\sum_{a_i \in S_j} a_i = b$? Given an instance of the 3-PARTITION problem, we construct an MUJSD system with distinct distance constraints as follows: $\mathbf{H} = \{H_1, H_2, \ldots, H_{3l}, H_{3l+1}\}$. $d_i = l(b+1) - a_i$ for $1 \le i \le 3l$, and $d_{3l+1} = b + 1$. $k_i = a_i - 1$ for $1 \le i \le 3l$, and $k_{3l+1} = l - 1$. Let c_i denote the distance constraint of the ith chain, i.e., the distance constraint between $J_{i,j-1}$ and J_{ij} for $1 \le j \le k_i$. $c_i = 1$ for $1 \le i \le 3l$, and $c_{3l+1} = b + 1$. It is easy to see that the transformation can be done in polynomial time.

Now, we show that the system constructed above is schedulable if and only if the instance of the 3-PARTITION problem has a solution. The total number of jobs in the system is $(3l+1) + \sum_{i=1}^{3l+1} k_i = (3l+1) + (lb - 3l) + (l-1) = l(b+1)$. Moreover, all jobs in the system, except job $J_{3l+1,l-1}$, have virtual deadlines less than or equal to $l(b+1) - 1$. That means job $J_{3l+1,l-1}$ cannot be scheduled before time $l(b+1) - 1$ and because of the restrictions of the deadline of job H_{3l+1} ($d_{3l+1} = b + 1$) and the distance constraint of the $(3l+1)$st chain ($c_{3l+1} = b + 1$), job $J_{3l+1,j}$ must be scheduled at time slot $(j+1)(b+1)$ for $0 \le j \le l - 1$. At this point, there are l disjoint time intervals available for the jobs in the first $3l$ chains, and each time interval has a length of exactly b.

Since the distance constraints of the first $3l$ chains are one, all jobs in each chain must be scheduled consecutively. This enforces the fact that all the jobs in each chain must be scheduled in the same time interval. Moreover, there are $k_i + 1 = a_i$ jobs in the ith chain for $1 \le i \le 3l$. It is easy to see that the system is schedulable if and only if the partition exists for the instance of the 3-PARTITION problem. This concludes our proof. □

The above strong NP-completeness result shows that it is unlikely to find an algorithm in time polynomial in the total number of jobs n for the MUJSD problem in which different chains can have different distance constraints. It also implies that the general MUJSD problem is NP-complete.

5.2. The bilevel tree UJSD problem. We have shown that the MUJSD problem with distinct distance constraints is NP-complete. With distinct distance constraints, even if we restrict the graph to a bilevel tree (not a bilevel chain tree) we can show that the problem is still NP-complete.

THEOREM 5.2. *The UJSD problem is NP-complete even if the graph is a bilevel tree.*

Proof. It is easy to see that the bilevel tree UJSD is in NP. To complete the proof, we shall reduce 3SAT [5] to it. The 3SAT problem is as follows: Given a set $\mathbf{U} = \{u_1, u_2, \ldots, u_n\}$ of n variables and a collection $\mathbf{C} = \{c_1, c_2, \ldots, c_m\}$ of m clauses over $\mathbf{U}$ such that $|c_i| = 3$ for $1 \leq i \leq m$, does there exist a satisfying truth assignment for $\mathbf{C}$? Given an instance of 3SAT we construct a bilevel tree UJSD problem in polynomial time as follows: Let $c_i = \{v_{i1}, v_{i2}, v_{i3}\}$ for $1 \leq i \leq m$. Define R as the dummy root of the bilevel tree. Also, define the head jobs as u_i and $\bar{u}_i$ for $1 \leq i \leq n$, and the tail jobs as w_{ij} for $1 \leq i \leq m$ and $1 \leq j \leq 3$. Each head job u_l or $\bar{u}_l$ has a deadline $2l$, i.e., there is a directed edge from the root R to u_l and another to $\bar{u}_l$, both with a deadline constraint $2l$. For each job w_{ij}, if $v_{ij} = u_l$ ($\bar{u}_l$) in 3SAT, then there is a precedence constraint from job u_l ($\bar{u}_l$) to job w_{ij} with a distance constraint $2n - 2l + 3i$.

To show that the collection $\mathbf{C}$ is satisfiable if and only if the constructed bilevel tree UJSD has a feasible schedule, let us note the following observations. The dummy job R is assumed to be scheduled at slot 0. All of the other jobs must be scheduled from slot 1 to slot $2n + 3m$. For all l, the deadlines of jobs u_l and $\bar{u}_l$ are both $2l$. Since jobs u_1 and $\bar{u}_1$ can be scheduled at slots 1 and 2, only one of them must be scheduled at slot 1 and the other must be scheduled at slot 2. Similarly, if the scheduling problem has a feasible schedule, one of the two jobs u_l and $\bar{u}_l$ must be scheduled at slot $2l - 1$ and the other must be scheduled at slot $2l$. Let's call slot $2l - 1$ the *falsity* slot of jobs u_l and $\bar{u}_l$ and call slot $2l$ the *truth* slot of jobs u_l and $\bar{u}_l$. The virtual deadline of job w_{ij}, whose predecessor is either u_l or $\bar{u}_l$, is $2l + 2n - 2l + 3i = 2n + 3i$, which is independent of j. Again, for the scheduling problem to have a feasible schedule, the three jobs w_{i1}, w_{i2}, and w_{i3} must be scheduled at the three slots $2n + 3i - 2$, $2n + 3i - 1$, and $2n + 3i$. Moreover, any one of the three w_{ij}'s can be scheduled at slot $2n + 3i$ if and only if its predecessor u_l (or $\bar{u}_l$) is scheduled at slot $2l$. From the above observations, it is easy to see that a feasible schedule exists if and only if, for each set of the three jobs $\{w_{i1}, w_{i2}, w_{i3}\}$, $1 \leq i \leq m$, at least one of their predecessors is scheduled at its truth slot. □

5.3. The multiprocessor problems. In all of the above discussions we only consider single processor scheduling. As we can expect, the problem becomes harder if there are multiple processors. The following lemma and theorem discuss multiprocessor scheduling with distance constraints.

DEFINITION. *A schedule for an MUJSD system on P processors is called* nonsplit *if all jobs in each chain are scheduled on the same processor; otherwise it is called a* split *schedule.*

For an MUJSD system with distance constraint $c = 1$, each job must be scheduled right after its immediate predecessor. However, two consecutive jobs need not be executed on the same processor. For the case in which $c = 1$, we call time t a *split point* of a schedule S if there is a chain that is split on different processors at time t in S.

LEMMA 5.3. *If an MUJSD system with distance constraint $c = 1$ is schedulable on P processors, then there exists a feasible nonsplit schedule for the system.*

Proof. Let S be a split schedule for the system. We now show, for $P = 2$, that S can be transformed to a nonsplit schedule. The proof for $P > 2$ can be shown in a similar way. Let t be the earliest split point in S. Suppose the ith chain is split at time t. Without loss of generality, assume that jobs $J_{i0}, J_{i1}, \ldots, J_{i,j-1}$ are scheduled at times $t-j, t-j+1, \ldots, t-1$, respectively, on processor 1; jobs $J_{ij}, J_{i,j+1}, \ldots, J_{i,j+l}$ are scheduled at times $t, t+1, \ldots, t+l$, respectively, on processor 2; and jobs (if any) $J_{i,j+l+1}, J_{i,j+l+2}, \ldots, J_q$ are scheduled after time $t+l+1$ on processor 1. Moreover, assume that jobs $Q_0, Q_1, \ldots, Q_l$ are scheduled at times $t, t+1, \ldots, t+l$, respectively, on processor 1. Then, we can move jobs $J_{ij}, J_{i,j+1}, \ldots, J_{i,j+l}$ from processor 2 to processor 1 and move jobs $Q_0, Q_1, \ldots, Q_l$ from processor 1 to processor 2 to get another feasible schedule S'. It is easy to see that the earliest split point t' of S' must be larger than the earliest split point t of S. Repeating the above process we will finally get a feasible nonsplit schedule, since the total completion time of schedule S is finite. □

THEOREM 5.4. *The multiprocessor MUJSD problem is (at least) NP-complete in the ordinary sense if the number of processors P is fixed, and NP-complete in the strong sense if P is arbitrary.*

Proof. If the distance constraint c is equal to 1 then the problem is equivalent to the traditional nonpreemptive multiprocessor scheduling problem, except that in the multiprocessor MUJSD problem a chain of jobs can be split and scheduled on more than one processor as long as the (temporal) distance between a job and its immediate predecessor is less than or equal to the distance constraint $c = 1$. However, by the above lemma, the split and nonsplit cases are equivalent for $c = 1$. Therefore, the theorem follows from the known results that the nonpreemptive multiprocessor scheduling problem is NP-complete in the ordinary sense if P is fixed and NP-complete in the strong sense if P is arbitrary [5]. □

6. Conclusions. Many real-time systems must enforce temporal distance constraints on some of their jobs. In this paper, we investigated the scheduling problem in which jobs have some relative temporal distance constraints and presented some basic results on the scheduling problems for jobs with distance constraints. We showed that some of the special cases of the unit-time JSD problem are NP-complete. However, we presented an $O(n^2)$ pseudopolynomial-time algorithm and an $O(m^2c^2)$ polynomial-time algorithm for the MUJSD problem with m chains, n total jobs, and a uniform distance constraint c in the MUJSD system. Some extensions of the MUJSD problem have also been studied.

Acknowledgment. The authors thank the anonymous referee for valuable comments and suggestions on an earlier manuscript of this paper.

REFERENCES

[1] D. ADOLPHSON AND T. C. HU, *Optimal linear ordering*, SIAM J. Appl. Math., 25 (1973), pp. 403–423.
[2] J. BLAZEWICZ, *Selected topics in scheduling theory*, Ann. Discrete Math., 31 (1987), pp. 1–60.
[3] M. R. GAREY, R. L. GRAHAM, D. S. JOHNSON, AND D. E. KNUTH, *Complexity results for bandwidth minimization*, SIAM J. Appl. Math., 34 (1978), pp. 477–495.
[4] M. R. GAREY AND D. S. JOHNSON, *Two-processor scheduling with start-times and deadlines*, SIAM J. Comput., 6 (1977), pp. 416–426.
[5] ———, *Computers and Intractability: A Guide to Theory of NP-Completeness*, W. H. Freeman, San Francisco, CA, 1979.
[6] R. L. GRAHAM, E. L. LAWER, J. K. LENSTRA, and A. H. G. RINNOOY KAN, *Optimization and approximation in deterministic sequencing and scheduling: A survey*, Ann. Discrete Math., 5 (1979), pp. 287–326.
[7] C.-C. HAN, *Scheduling real-time computations with temporal distance and separation constraints and with extended deadlines*, Ph.D. thesis, Tech. report UIUCDCS-R-92-1748, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992.
[8] C.-C. HAN AND K.-J. LIN, *Scheduling jobs with temporal consistency constraints*, in Proc. 6th IEEE Workshop on Real-Time Operating Systems and Software, Pittsburgh, PA, 1989, pp. 18–23.
[9] ———, *Scheduling distance-constrained real-time tasks*, in Proc. 13th IEEE Real-Time Systems Symposium, Phoenix, AZ, 1992, pp. 300–308.
[10] ———, *Scheduling real-time computations with separation constraints*, Inform. Process. Lett. 42 (1992), pp. 61–66.
[11] E. L. LAWLER, J. K. LENSTRA, AND A. H. G. RINNOOY KAN, *Recent developments in deterministic sequencing and scheduling: A survey*, in Deterministic and Stochastic Scheduling, M. A. H. Dempster, J. K. Lenstra, and A. H. G. Rinnooy Kan, eds., D. Reidel, Dordrecht, the Netherlands, 1982, pp. 285–298.
[12] J. K. LENSTRA AND A. H. G. RINNOOY KAN, *Scheduling theory since 1981: An annotated bibliography*, in Combinatorial Optimization: Annotated Bibliographies, M. O'hEigeartaigh, J. K. Lenstra, and A. H. G. Rinnooy Kan, eds., John Wiley, Chichester, 1984.
[13] C. H. PAPADIMITRIOU, *The NP-completeness of the bandwidth minimization problem*, Computing, 16 (1976), pp. 263–270.
[14] B. SIMONS, *A fast algorithm for single processor scheduling*, in Proc. 19th Symposium on Foundations of Computer Science, Long Beach, CA, IEEE Computer Society, 1978, pp. 246–252.

SIAM J. COMPUT.
Vol. 24, No. 5, pp. 1122–1139, October 1995

012

ON THE APPROXIMATION OF SHORTEST COMMON SUPERSEQUENCES AND LONGEST COMMON SUBSEQUENCES*

TAO JIANG† AND MING LI‡

Abstract. The problems of finding shortest common supersequences (SCS) and longest common subsequences (LCS) are two well-known **NP**-hard problems that have applications in many areas, including computational molecular biology, data compression, robot motion planning, and scheduling, text editing, etc. A lot of fruitless effort has been spent in searching for good approximation algorithms for these problems. In this paper, we show that these problems are inherently hard to approximate in the worst case. In particular, we prove that (i) SCS does not have a polynomial-time linear approximation algorithm unless $\mathbf{P} = \mathbf{NP}$; (ii) There exists a constant $\delta > 0$ such that, if SCS has a polynomial-time approximation algorithm with ratio $\log^{\delta} n$, where n is the number of input sequences, then **NP** is contained in $\mathbf{DTIME}(2^{\text{polylog } n})$; (iii) There exists a constant $\delta > 0$ such that, if LCS has a polynomial-time approximation algorithm with performance ratio n^{δ}, then $\mathbf{P} = \mathbf{NP}$. The proofs utilize the recent results of Arora et al. [*Proc. 23rd IEEE Symposium on Foundations of Computer Science*, 1992, pp. 14–23] on the complexity of approximation problems.

In the second part of the paper, we introduce a new method for analyzing the average-case performance of algorithms for sequences, based on Kolmogorov complexity. Despite the above nonapproximability results, we show that near optimal solutions for both SCS and LCS can be found on the average. More precisely, consider a fixed alphabet Σ and suppose that the input sequences are generated randomly according to the uniform probability distribution and are of the same length n. Moreover, assume that the number of input sequences is polynomial in n. Then, there are simple greedy algorithms which approximate SCS and LCS with expected additive errors $O(n^{0.707})$ and $O(n^{1/2+\epsilon})$ for any $\epsilon > 0$, respectively.

Incidentally, our analyses also provide tight upper and lower bounds on the expected LCS and SCS lengths for a set of random sequences solving a generalization of another well-known open question on the expected LCS length for two random sequences [K. Alexander, *The rate of convergence of the mean length of the longest common subsequence*, 1992, manuscript], [V. Chvatal and D. Sankoff, *J. Appl. Probab.*, 12 (1975), pp. 306–315], [D. Sankoff and J. Kruskall, eds., *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison–Wesley, Reading, MA, 1983].

Key words. shortest common supersequence, longest common subsequence, approximation algorithm, **NP**-hardness, average-case analysis, random sequence

AMS subject classifications. 68Q20, 68Q25

1. Introduction.

For two sequences $s = s_1 \dots s_m$ and $t = t_1 \dots t_n$, we say that s is a *subsequence* of t and, equivalently, t is a *supersequence* of s, if for some $i_1 < \cdots < i_m$, $s_j = t_{i_j}$. Given a finite set of sequences S, a *shortest common supersequence* (SCS) of S is a shortest possible sequence s such that each sequence in S is a subsequence of s. A *longest common subsequence* (LCS) of S is a longest possible sequence s such that each sequence in S is a supersequence of s.

These problems arise naturally in many practical situations. Researchers in many different areas have been trying for years to obtain partial solutions: dynamic programming, when the number of sequences is constant, or, ad hoc algorithms when we do not care about absolute optimal solutions.

In artificial intelligence (specifically, planning), the actions of a robot (and human for that matter) need to be planned. A robot usually has many goals to achieve. To achieve each goal the robot sometimes needs to perform a (linearly ordered) sequence of operations,

*Received by the editors October 9, 1992; accepted for publication (in revised form) May 31, 1994.

†Department of Computer Science and Systems, McMaster University, Hamilton, Ontario L8S 4K1, Canada (jiang@maccs.mcmaster.ca). The research of this author was supported in part by Natural Sciences and Engineering Research Council of Canada operating grant OGP0046613 and the Canadian Genome Analysis and Technology program.

‡Department of Computer Science, University of Waterloo, Waterloo, Ontario N3L 3G1, Canada (mli@math.uwaterloo.ca). The research of this author was supported in part by Natural Sciences and Engineering Research Council of Canada operating grant OGP0046506, the Information Technology Research Center, and the Canadian Genome Analysis and Technology program.

where identical operations in different sequences may be factored out and only performed once. Optimally merging these sequences of actions of the robot implies efficiency. Practical examples include robotic assembly lines [8] and metal cutting in the domain of automated manufacturing [10], [15]. The essence of solutions to such problems are good approximation algorithms for the SCS problem. The SCS problem also has applications in text editing [24], data compression [27], and query optimization in database systems [25].

In molecular biology, an LCS (of some DNA sequences) is commonly used as a measure of similarity in the comparison of biological sequences [6]. Efficient algorithms for finding an LCS could be very useful. Many papers in molecular biology have been written on this issue. We refer the readers to [6], [7], and [26]. Other applications of the LCS problem include the widely used *diff* file comparison program [1], data compression, and syntactic pattern recognition [19].

For a long time, it has been well known that the SCS and LCS problems (even on a binary alphabet) are all **NP**-hard [9], [21], [23]. For n sequences of length m, it is known that, by dynamic programming techniques, both SCS and LCS problems can be solved in $O(m^n)$ time; this result is independently a result of many authors in computer science (e.g., [11], [29]) and biology. However, since the parameter m is usually extremely large in practice (e.g., in computer text editing and DNA/RNA sequence comparison), the time requirement $O(m^n)$ is intolerable even for small to moderate n. There have been attempts to speed up the dynamic programming solution for LCS [12], [13]. The improved algorithms still run in $O(m^n)$ time in the worst case.

In the very first paper [21] that proves the **NP**-hardness of the SCS and LCS problems, Maier already asks for approximation algorithms. For the past many years, various groups of people in the diverse fields of artificial intelligence, theoretical computer science, and biology have looked for heuristic algorithms to approximate the SCS and LCS problems, but so far no polynomial time algorithms with guaranteed approximation bounds have been found.

In this paper, we show that it is indeed not surprising that the search for good approximation algorithms has been fruitless, because good approximations of these problems would imply that $\mathbf{P} = \mathbf{NP}$. Specifically, we show the following:

1. No polynomial-time algorithm can achieve a constant approximation ratio for any constant for the SCS problem unless $\mathbf{P} = \mathbf{NP}$. (The approximation ratio is the worst-case ratio between the approximate solution and the optimal solution.)
2. There exists a constant $\delta > 0$ such that, if SCS has a polynomial-time approximation algorithm with ratio $\log^{\delta} n$, where n is the number of input sequences, then **NP** is contained in $\mathbf{DTIME}(2^{\text{polylog } n})$.
3. There exists a constant $\delta > 0$ such that, if the LCS problem has a polynomial-time approximation algorithm with performance ratio n^{δ}, then $\mathbf{P} = \mathbf{NP}$.

The above results assume an unbounded alphabet. Our proofs utilize the recent results of Arora et al. on the complexity of approximation problems [3]. An overview of the results in [3] that are of special interest to us will be given in the next section.

On the other hand, one should not be too discouraged by the nonapproximability results above. Many heuristic algorithms for SCS and LCS seem to work well in practice. They are usually greedy-style algorithms and run very fast. These algorithms cannot always guarantee an optimal solution or even an approximately good solution, but they produce a satisfactory solution in most cases. In this paper, we try to provide a (partial) theoretical explanation by proving that both SCS and LCS can be indeed very well approximated *on the average*. Consider the SCS and LCS problems on a fixed alphabet Σ. Suppose that the input sequences are of length n, the number of input sequences is polynomial in n, and all sequences are *equally likely* and mutually independent. We show that some simple greedy algorithms approximate SCS and LCS with *expected additive errors* $O(n^{0.707})$ and $O(n^{1/2+\epsilon})$ for any $\epsilon > 0$, respectively.

(The additive error is the difference between the approximate solution and optimal solution.) The algorithm for SCS is actually interesting and perhaps useful in practice, although the algorithm for LCS is somewhat trivial and impractical. Our analyses are based on a new average-case analysis method using Kolmogorov complexity.

It also turns out that such analyses enable us also to obtain a tight bound on the expected length of an LCS or an SCS of random sequences. Our results show that, over a fixed alphabet of size k, the expected length of an LCS (or an SCS) for n random sequences of length n is $\frac{n}{k} \pm n^{1/2+\epsilon}$ for any $\epsilon > 0$ (or $\frac{(k+1)n}{2} \pm O(n^{0.707})$, respectively). In contrast, the tight bound on the expected LCS length for two random sequences is a well-known open question in statistics [2], [5], [24].

In §2, we review the recent surprisingly fast development in the complexity theory of approximation. The hardness of approximating an SCS or an LCS is shown in §3. We analyze the average-case performance of some greedy algorithms in §4. Some concluding remarks are given in §5.

2. Recent works on the complexity of approximation. Designing efficient approximation algorithms with good performance guarantees is not an easy task. This is a result of the fact that the approximation of a large number of optimization problems is intractable. On the other hand, proving the intractability of an approximation problem can also be hard, essentially because the approximability properties are generally not preserved in a conventional polynomial-time reduction [9]. Nevertheless, there have been some very significant developments in the last five years. We only discuss the work that will be needed for our results.

In 1988, Papadimitriou and Yannakakis defined a special reduction that preserves certain approximability properties [22]. Using this reduction, based on Fagin's syntactic definition of the class **NP**, they introduced a class of natural optimization problems, **MAX SNP**, which includes the vertex cover and independent set problems on bounded-degree graphs, max cut, various versions of maximum satisfiability, etc. It is known that every problem in this class can be approximated within *some* constant factor, and a polynomial-time approximation scheme (PTAS) for any **MAX SNP**-complete problem would imply one for every other problem in the class. (A problem has a PTAS if, for every fixed $\epsilon > 0$, the problem can be approximated within factor $1 + \epsilon$ in polynomial time.)

Recently, Arora et al. made some significant progress in the theory of interactive proofs [3]. As an application of their results, they showed that if any **MAX SNP**-hard problem has a PTAS, then $\mathbf{P} = \mathbf{NP}$, thus confirming the common belief that no **MAX SNP**-hard problem has a PTAS. Their results also show that, unless $\mathbf{P} = \mathbf{NP}$, the largest clique problem does not have a polynomial-time approximation algorithm with performance ratio n^{δ} for some constant δ. Using these results and the graph product technique, Karger, Motwani, and Ramkumar are able to prove that longest paths cannot be approximated within any constant factor [14]. Very recently, Lund and Yannakakis showed that graph coloring cannot be approximated with ratio n^{ϵ} for some $\epsilon > 0$ and set covering cannot be approximated with ratio $c \log n$ for any $c < \frac{1}{4}$ [20].

Before we leave this section, we recall the definition of the special reduction introduced by Papadimitriou and Yannakakis [22], used to show the **MAX SNP**-hardness of a problem. Suppose that Π and Π' are two optimization (i.e., maximization or minimization) problems. We say that Π *L-reduces* (*linearly reduces*) to Π' if there are two polynomial-time algorithms f and g and constants $\alpha, \beta > 0$ such that, for any instance I of Π,

1. $\mathrm{OPT}((I)) \leq \alpha \cdot \mathrm{OPT}(I)$;
2. given any solution of $f(I)$ with weight w', algorithm g produces in polynomial time a solution of I with weight w satisfying $|w - \mathrm{OPT}(I)| \leq \beta |w' - \mathrm{OPT}(f(I))|$.

The following are two simple facts concerning L-reductions. First, the composition of two L-reductions is also an L-reduction. Second, if problem Π L-reduces to problem Π' and Π' can be approximated in polynomial time within a factor of $1 + \epsilon$, then Π can be approximated within factor $1 + \alpha\beta\epsilon$. In particular, if Π' has a PTAS, then so does Π.

A problem is **MAX SNP**-hard if every problem in **MAX SNP** can be L-reduced to it. Thus, by the result of Arora et al., a **MAX SNP**-hard problem does not have a PTAS unless $\mathbf{P} = \mathbf{NP}$.

3. Nonapproximability of SCS and LCS. In this section, we show that there do not exist polynomial-time approximation algorithms for SCS and LCS with *good* performance ratios. The proof for LCS is a direct reduction from the largest clique problem. The proof for SCS is more involved. We first show that a restricted version of SCS, in which every input sequence is of length 2 and every letter of the alphabet appears at most three times in the input sequences, is **MAX SNP**-hard. Thus this restricted version of SCS does not have a PTAS, assuming that $\mathbf{P} \neq \mathbf{NP}$. Then we define the product of sets of sequences and relate the SCS of such a product to the SCS's of the components constituting the product. Finally, we demonstrate that a polynomial-time constant ratio approximation algorithm for SCS would imply a PTAS for the restricted SCS by blowing up instances using the product of sets of sequences.

3.1. Approximating LCS is hard.

THEOREM 3.1. *There exists a constant $\delta > 0$ such that, if the LCS problem has a polynomial time-approximation algorithm with performance ratio n^δ, where n is the number of input sequences, then* $\mathbf{P} = \mathbf{NP}$.

Proof. We reduce the largest clique problem to LCS. Let $G = (V, E)$ be a graph and $V = \{v_1, \ldots, v_n\}$ be the vertex set. Our alphabet Σ is chosen to be V.

Consider a vertex v_i and suppose that v_i is adjacent to vertices $v_{i_1}, \ldots, v_{i_q}$, where $i_1 < \cdots < i_q$. For convenience, let $i_0 = 0$ and $i_{q+1} = n + 1$. Let p be the unique index such that $0 \leq p \leq q$ and $i_p < i < i_{p+1}$. We include the following two sequences:

$$x_i = v_{i_1} \ldots v_{i_p} v_i v_1 \ldots v_{i-1} v_{i+1} \ldots v_n,$$
$$x_i' = v_1 \ldots v_{i-1} v_{i+1} \ldots v_n v_i v_{i_{p+1}} \ldots v_{i_q}.$$

Let $S = \{x_i, x_i' | 1 \leq i \leq n\}$.

LEMMA 3.2. *The graph G has a clique of size k if and only if the set S has a common subsequence of length k for any k.*

Proof. The "only if" part is clear. To prove the "if" part, let y be a common subsequence for S. If v_i appears in y, then the sequence x_i makes sure that all vertices on the left of v_i in y are adjacent to v_i in G and, similarly, the sequence x_i' ensures that that all vertices on the right of v_i in y are adjacent to v_i in G. Thus the vertices appearing in y actually form a clique of G. □

The proof is completed by recalling the result of Arora et al. which states that, unless $\mathbf{P} = \mathbf{NP}$, the largest clique problem does not have a polynomial-time approximation algorithm with performance ratio n^δ on graphs with n vertices for some constant $\delta > 0$ [3]. □

The above result shows that for some constant $\delta > 0$, there is no algorithm that, given a set S of n sequences, will find a common subsequence of length at least $\mathrm{OPT}(S)/n^\delta$ in polynomial time. But the question of whether there might be some other $\delta < 1$, such that one can find a common subsequence of length at least $\mathrm{OPT}(S)/n^\delta$ in polynomial time, still remains open. We conjecture that the answer is negative.

It is also natural to measure the performance of an approximation algorithm for LCS in terms of the size of the alphabet or the maximum length of its input sequences. Clearly Theorem 3.1 holds when n is replaced by these parameters.

We now consider the approximation of LCS on a fixed alphabet. Let the alphabet $\Sigma = \{a_1, \ldots, a_k\}$. It is trivial to show that LCS on Σ can be approximated with ratio k.

THEOREM 3.3. *For any set S of sequences from Σ, the algorithm, Long-Run, finds a common subsequence for S of length at least* $\mathrm{OPT}(S)/k$.

ALGORITHM LONG-RUN.

Find maximum m such that a^m is a common subsequence of all input sequences for some $a \in \Sigma$. Output a^m as the approximation of LCS.

The question of whether LCS on a bounded alphabet is **MAX SNP**-hard remains open. Although we believe that the answer is "yes," even when the alphabet is binary, we have not been able to establish an L-reduction from any known **MAX SNP**-hard problem. Observe that Maier's construction for the **NP**-hardness of LCS on a bounded alphabet [21] does not constitute an L-reduction. In his construction, an instance G of vertex cover is mapped to an instance S of LCS (or SCS) with the property that $\mathrm{OPT}(S)$ is at least quadratic in $\mathrm{OPT}(G)$.

Conjecture. LCS on a binary alphabet is **MAX SNP**-hard.

3.2. Restricted versions of SCS and MAX SNP-hardness. Maier proved the **NP**-hardness of SCS on an unbounded alphabet by reducing the vertex cover problem to SCS [21]. For any graph G of n vertices and m edges, the construction guarantees that G has a vertex cover of size t if and only the constructed instance of SCS has a common supersequence of length $2n + 6m + 8\max\{n, m\} + t$. It is easy to see that the reduction is actually linear if the graph G is of bounded degree. Since the vertex cover problem on bounded-degree graphs is **MAX SNP**-hard, so is SCS.

Let $\mathrm{SCS}(l, r)$ denote the restricted version of SCS in which each input sequence is of length l and each letter appears at most r times totally in all sequences. Such restricted problems have been recently studied by Timkovskii [28]. We will need the version SCS(2, 3) later to prove that SCS cannot be linearly approximated. It is known that SCS(2, 2) can be solved in polynomial time and SCS(2, 3) is **NP**-hard [28]. Obviously, $\mathrm{SCS}(l, r)$ can be approximated with ratio r. This is true because each letter appears only r times in total in an instance of $\mathrm{SCS}(l, r)$. Thus a plain concatenation already achieves an approximation ratio r.

THEOREM 3.4. SCS(2, 3) *does not have a PTAS unless* $\mathbf{P} = \mathbf{NP}$.

Proof. A polynomial-time reduction from the feedback vertex set problem on bounded-degree digraphs [9] to SCS(2, 3) is given in [28]. Let $G = (V, E)$ be a digraph of degree 3. The reduction defines a set $S = \{uv | (u, v) \in E\}$. Clearly, S is an instance of SCS(2, 3). It is shown that G has a feedback vertex set of size t if and only if S has a common supersequence of length $|V| + t$.

It is easy to L-reduce vertex cover on bounded-degree graphs to feedback vertex set on bounded-degree graphs by replacing each edge in the instance of vertex cover with a directed cycle. The reduction from feedback vertex set to SCS(2, 3) is actually linear for the digraphs resulting from this construction, because the optimal feedback vertex set for these graphs is linear in $|V|$. Since the composition of two L-reductions is an L-reduction, we have an L-reduction from the vertex cover problem on bounded degree graphs to SCS(2, 3). The theorem follows from the fact that vertex cover on bounded degree graphs is **MAX SNP**-hard. □

The status of the complexity of approximating SCS on a fixed alphabet is quite similar to that for LCS. We can show that SCS on a fixed alphabet has a trivial constant ratio approximation. But we do not know if the problem is **MAX SNP**-hard. Again, observe that the reduction in Maier's original proof of the **NP**-hardness for SCS on a bounded alphabet is not linear.

THEOREM 3.5. *Let Σ be an alphabet of k letters. For any set S of sequences from Σ, we can find a common supersequence for S of length at most $k \cdot \mathrm{OPT}(S)$ in polynomial time.*

Proof. Let $l_{\max}$ be the maximum length of input sequences in S. Then $(a_1 \ldots a_k)^{l_{\max}}$ is a common supersequence for S satisfying the length requirement. □

Conjecture. SCS on a binary alphabet is **MAX SNP**-hard.

3.3. The product of sets of sequences. First, we extend the operation "concatenation" to sets of sequences. Let X and Y be two sets of sequences. Define the concatenation of X, Y, denoted $X \cdot Y$, as the set $\{x \cdot y | x \in X, y \in Y\}$. For example, if $X = \{abab, aabb\}$ and $Y = \{123, 231, 312\}$, then

$$X \cdot Y = \{abab123, aabb123, abab231, aabb231, abab312, aabb312\}.$$

The following lemma is quite useful in our construction.

LEMMA 3.6. *Let $X = X_1 \cdot X_2 \ldots X_n$. Suppose that y is a supersequence for X. Then there exist $y_1, y_2, \ldots, y_n$ such that $y = y_1 \cdot y_2 \ldots y_n$ and each y_i is a supersequence for X_i, $1 \le i \le n$.*

Proof. Let $X_1 = \{x_1, \ldots, x_p\}$ and $X' = X_2 \ldots X_n = \{x'_1, \ldots, x'_q\}$. Fix a sequence x'_i in X'. Since the supersequence y contains $x_1 \cdot x'_i, \ldots, x_p \cdot x'_i$ as subsequences, it must contain a subsequence $y_{1,i} \cdot x'_i$, where $y_{1,i}$ is a supersequence for X_1. To see this, just consider the positions of the first letter of x'_i in the subsequences $x_1 \cdot x'_i, \ldots, x_p \cdot x'_i$ of y. The rightmost occurrence gives $y_{1,i}$. See Fig. 1. In the figure, the $*$'s represent sequences $x_1, \ldots, x_p$ and the #'s represent the sequences $x'_1, \ldots, x'_q$. The sequences are aligned according to their relative positions in y.

Thus, for each $1 \le i \le q$, we have a sequence $y_{1,i}$ such that $y_{1,i}$ is a supersequence for X_1 and $y_{1,i} \cdot x'_i$ is a subsequence of y. Now we consider the positions of the last letter of $y_{1,1}, \ldots, y_{1,q}$ in the subsequences $y_{1,1} \cdot x'_1, \ldots, y_{1,q} \cdot x'_q$ of y and partition y at the leftmost such position. Let the left part of y be y_1 and the right part be y'. Then, clearly, y_1 is a supersequence for X_1 and y' is a supersequence for X'. See Fig. 2. In this figure, the $*$'s now represent sequences $y_{1,1}, \ldots, y_{1,q}$.

We can do this recursively on y' and Y' to obtain $y_2, \ldots, y_n$. □

We are now ready to define the product of sets of sequences. The symbol $\times$ will be used to denote the product operation. We start by defining the product of single letters. Let Σ and Σ' be two alphabets and $a \in \Sigma, b \in \Sigma'$ be two letters. The product of a and b is simply the composite letter $(a, b) \in \Sigma \times \Sigma'$. The product of a sequence $x = a_1 \cdot a_2 \ldots a_n$ and a letter b is $a_1 \times b \cdot a_2 \times b \ldots a_n \times b$. The product of a set $X = \{x_1, x_2, \ldots, x_n\}$ of sequences and a letter a is the set $\{x_1 \times a, x_2 \times a, \ldots, x_n \times a\}$. The product of a set X of sequences and a sequence $y = a_1 \cdot a_2 \ldots a_n$ is the set $X \times a_1 \cdot X \times a_2 \ldots X \times a_n$. Finally, let X and $Y = \{y_1, y_2, \ldots, y_n\}$ be two sets of sequences. The product of X and Y is the set $\cup_{i=1}^{n} X \times y_i$. For example, if $X = \{aabb, abab\}$ and $Y = \{121, 212\}$, then $X \times Y$ contains the 16 sequences in Fig. 3.

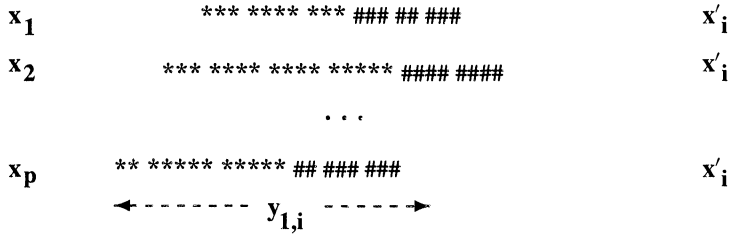


FIG. 1. *Finding $y_{1,i}$ in the supersequence y.*

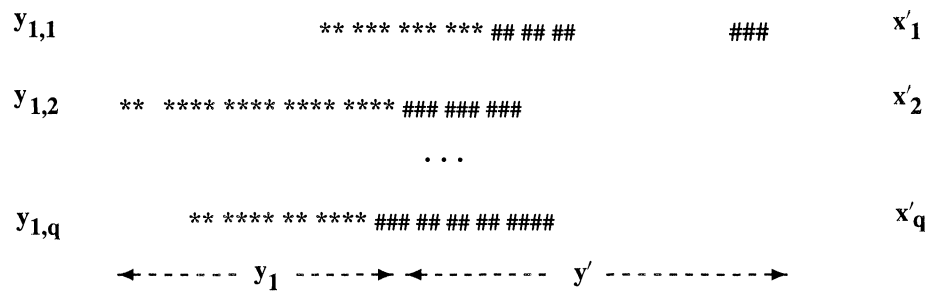


FIG. 2. *Finding y_1 in the supersequence y.*

$(a, 1)(a, 1)(b, 1)(b, 1)(a, 2)(a, 2)(b, 2)(b, 2)(a, 1)(a, 1)(b, 1)(b, 1)$
$(a, 1)(a, 1)(b, 1)(b, 1)(a, 2)(a, 2)(b, 2)(b, 2)(a, 1)(b, 1)(a, 1)(b, 1)$
$(a, 1)(a, 1)(b, 1)(b, 1)(a, 2)(b, 2)(a, 2)(b, 2)(a, 1)(a, 1)(b, 1)(b, 1)$
$(a, 1)(a, 1)(b, 1)(b, 1)(a, 2)(b, 2)(a, 2)(b, 2)(a, 1)(b, 1)(a, 1)(b, 1)$
$(a, 1)(b, 1)(a, 1)(b, 1)(a, 2)(a, 2)(b, 2)(b, 2)(a, 1)(a, 1)(b, 1)(b, 1)$
$(a, 1)(b, 1)(a, 1)(b, 1)(a, 2)(a, 2)(b, 2)(b, 2)(a, 1)(b, 1)(a, 1)(b, 1)$
$(a, 1)(b, 1)(a, 1)(b, 1)(a, 2)(b, 2)(a, 2)(b, 2)(a, 1)(a, 1)(b, 1)(b, 1)$
$(a, 1)(b, 1)(a, 1)(b, 1)(a, 2)(b, 2)(a, 2)(b, 2)(a, 1)(b, 1)(a, 1)(b, 1)$

$(a, 2)(a, 2)(b, 2)(b, 2)(a, 1)(a, 1)(b, 1)(b, 1)(a, 2)(a, 2)(b, 2)(b, 2)$
$(a, 2)(a, 2)(b, 2)(b, 2)(a, 1)(a, 1)(b, 1)(b, 1)(a, 2)(b, 2)(a, 2)(b, 2)$
$(a, 2)(a, 2)(b, 2)(b, 2)(a, 1)(b, 1)(a, 1)(b, 1)(a, 2)(a, 2)(b, 2)(b, 2)$
$(a, 2)(a, 2)(b, 2)(b, 2)(a, 1)(b, 1)(a, 1)(b, 1)(a, 2)(b, 2)(a, 2)(b, 2)$
$(a, 2)(b, 2)(a, 2)(b, 2)(a, 1)(a, 1)(b, 1)(b, 1)(a, 2)(a, 2)(b, 2)(b, 2)$
$(a, 2)(b, 2)(a, 2)(b, 2)(a, 1)(a, 1)(b, 1)(b, 1)(a, 2)(b, 2)(a, 2)(b, 2)$
$(a, 2)(b, 2)(a, 2)(b, 2)(a, 1)(b, 1)(a, 1)(b, 1)(a, 2)(a, 2)(b, 2)(b, 2)$
$(a, 2)(b, 2)(a, 2)(b, 2)(a, 1)(b, 1)(a, 1)(b, 1)(a, 2)(b, 2)(a, 2)(b, 2)$

FIG. 3. *The product of* $\{aabb, abab\}$ *and* $\{121, 212\}$.

Note that if each sequence in X has length l_1 and each sequence in Y has length l_2, then $X \times Y$ contains $|Y| \cdot |X|^{l_2}$ sequences of length $l_1 \cdot l_2$. Thus, $X \times Y$ does not have polynomial size in general.

In this paper, we will only be interested in products in which the second operand has a special property. Let Y be a set of sequences with the following property: each sequence is of even length and every letter at an even position is *unique* in Y, i.e., the letter appears only once (totally) in all sequences in Y. We will refer to such unique letters as *delimiters*. The following lemma relates the SCS of a product to the SCS's of its operands and is crucial to §3.4.

LEMMA 3.7. *Let X and Y be sets of sequences. Suppose that Y has the above special property. Then* $\mathrm{OPT}(X \times Y) = \mathrm{OPT}(X) \cdot \mathrm{OPT}(Y)$. *Moreover, given a supersequence for $X \times Y$ of length l, we can find in time polynomial in $|X \times Y|$ supersequences for X and Y of lengths l_1 and l_2, respectively, such that $l_1 \cdot l_2 \leq l$.*

Proof. Clearly, $\mathrm{OPT}(X \times Y) \leq \mathrm{OPT}(X) \cdot \mathrm{OPT}(Y)$. Suppose that z is a supersequence for $X \times Y$ of length l. We show how to find supersequences for X and Y of lengths l_1 and l_2, respectively, such that $l_1 \cdot l_2 \leq l$ in polynomial time. Let Σ be the alphabet corresponding to Y. For each letter $a \in \Sigma$, we call the product $X \times a$ an *a component*. The letters are divided into delimiters and nondelimiters. For convenience, call the nondelimiters *normal* letters. Our basic idea is to rearrange the supersequence z without increasing length such that each component appears in a consecutive region. Then we can "extract" the components and identify the desired supersequences for X and Y.

Using Lemma 3.6, we can extract from z a supersequence for each delimiter component as follows. Let $y = a_1 a_2 \ldots a_n$ be a sequence in Y. Consider the delimiter a_2. Since z is a supersequence for the product $X \times y = X \times a_1 \cdot X \times a_2 \ldots X \times a_n$, there must be $z_1, z_2, \ldots, z_n$ such that z_i is a supersequence for $X \times a_i$ and $z = z_1 \cdot z_2 \ldots z_n$. Now we look at z_2 and concentrate on the sequences in the a_2 component. We can rearrange z_2 such that the letters appearing in the a_2 component form a consecutive block by shifting them to the right appropriately. Denote the new sequence z_2'. Since a_2 is unique in Y, $z_1 \cdot z_2' \ldots z_n$ is also a supersequence for $X \times Y$. This way we have extracted a supersequence for the a_2 component. Supersequences for other delimiter components can be extracted similarly. Note that the above does not increase the length of the whole supersequence.

Call the final supersequence after the above process z'. So z' has the form $u_1 \cdot v_1 \cdot u_2 \cdot v_2 \ldots$, where each v_i is a supersequence for some delimiter component and u_i is a sequence of letters from some normal components. We can easily rearrange z' so that each u_i actually becomes either nil or a supersequence for some normal component by shifting the normal component letters to the rightmost possible position (stopped only by some relevant delimiter block). Thus we now have a supersequence of the form $u_1' \cdot v_1 \cdot u_2' \cdot v_2 \ldots$, where each u_i' is either nil or a supersequence for some normal component. Now the pattern $u_1' \cdot v_1 \cdot u_2' \cdot v_2 \ldots$, naturally defines a supersequence for Y. Let l_1 be the minimum length of the supersequences for the components (and thus X) and l_2 be the length of the supersequence for Y. Then $l_1 \cdot l_2 \leq l$. □

3.4. SCS has no linear approximation algorithms. The basic idea is to use the product operation to blow up a given instance of SCS. However, the product of sets of sequences cannot be performed in polynomial time unless the the second operand contains sequences of bounded length. Thus we consider the restricted version SCS(2, 3). A nice thing about SCS(2, 3) is the fact that for any instance S of SCS(2, 3), the total length of the sequences in S is at most $3 \cdot \mathrm{OPT}(S)$. Thus, we can insert unique delimiters into the sequences as required in §3.3 without affecting the **MAX SNP**-hardness. So, let SCS(2, 3)′ denote the version whose instances are obtained from instances of SCS(2, 3) by inserting unique delimiters. Let S be an instance of SCS(2, 3) of total length n and S' be the corresponding instance of SCS(2, 3)′. It is easy to see that S has a common supersequence of length t if and only if S' has a common supersequence of length $n + t$. Since $t = \theta(n)$, this forms an L-reduction from SCS(2, 3) to SCS(2, 3)′, and hence SCS(2, 3)′ is also **MAX SNP**-hard.

For any set S, S^k denotes the product of k S's. The next lemma follows from Lemma 3.7.

LEMMA 3.8. *Let $k \geq 1$ be a fixed integer. For any instance S of* SCS(2, 3)′, $\mathrm{OPT}(S^k) = \mathrm{OPT}(S)^k$. *Moreover, given a supersequence for S^k of length l, we can find in polynomial time a supersequence for S of length $l^{1/k}$.*

Observe that if $|S| = n$, then $|S^k| = n^{O(4^k)}$, since each sequence in S has length 4. Now we can prove the main result in this section.

THEOREM 3.9. (i) *There does not exist a polynomial-time linear approximation algorithm for SCS unless* $\mathbf{P} = \mathbf{NP}$. (ii) *There exists a constant* $\delta > 0$ *such that, if SCS has a polynomial-time approximation algorithm with ratio* $\log^{\delta} n$, *where* n *is the number of input sequences, then* **NP** *is contained in* $\mathbf{DTIME}(2^{\text{polylog } n})$.

Proof. We only prove (i). The proof of (ii) is similar. The idea is to show that if SCS has a polynomial-time linear approximation algorithm, then SCS(2, 3)′ has a PTAS. Suppose that SCS has a polynomial-time approximation algorithm with performance ratio α. For any given $\epsilon > 0$, let $k = \lceil \log_{1+\epsilon} \alpha \rceil$. Then, by Lemma 3.8, we have an approximation algorithm for SCS(2, 3)′ with ratio $\alpha^{1/k} \leq 1 + \epsilon$. The algorithm runs in time $n^{O(4^k)}$, thus it is polynomial in n. This implies a PTAS for SCS(2, 3)′. □

It is interesting to note that our nonapproximability result for SCS is weaker than that of LCS (and the longest path problem in [14]). It seems to require new techniques to prove a stronger lower bound. The growth rate of $n^{O(4^k)}$ is too high. This is essentially a result of the way we define the product of sets of sequences. If we can find a better way of taking products and lower the rate to something like $n^{O(k)}$, then the bound in (ii) can be strengthened to $2^{\log^{\delta} n}$ for any $\delta < 1$, as shown in [14] for the longest path problem.

4. Algorithms with good average-case performance. We have seen that the LCS and SCS problems are not only **NP**-hard to solve exactly, but also **NP**-hard to approximate. The approximation of these problems restricted to fixed alphabets also seems to be hard. In this section, we consider the average-case performance of some simple greedy algorithms for LCS and SCS and prove that these algorithms can find a *nearly optimal* solution in *almost all the cases*, assuming that all sequences are equally likely and the sequences are independent of each other. Note that our probability model may not be realistic, because in practice the sequences are usually *related* to each other and thus are not independent.

From now on, let $\Sigma = \{a_1, \ldots, a_k\}$ be a fixed alphabet of size k. For convenience, we will assume that the input is always n sequences over Σ, each of length n, although our results actually hold when the number of input sequences is polynomial in n. We prove that some remarkably simple greedy algorithms can approximate LCS and SCS with minor expected additive errors. Some of the technical results are actually quite interesting in their own right. They give tight bounds on the expected length of an LCS or SCS of n random sequences of length n. It turns out that Kolmogorov complexity is a convenient and crucial tool for our analyses.

4.1. Kolmogorov complexity. Kolmogorov complexity has been used in [8] as an effective method for analyzing the average-case performance of some algorithms for the SCS problem. It was also used recently by Munro (see [17]) to obtain the average-case complexity of Heapsort, solving a long-standing open question. Here we use Kolmogorov complexity as a tool for analyzing some combinatorial properties of random sequences which result in tight upper and lower bounds on the average-case performance of some algorithms for LCS and SCS problems. One can compare the use of Kolmogorov complexity with the probabilistic method. In a sense, taking a Kolmogorov random string as the input is like taking an expectation as in the second moment method. A Kolmogorov random input has the random string properties, yet these properties hold with certainty rather than with some (high) probability. This fact greatly simplifies many proofs. To make this paper self contained, we briefly review the definition and some properties of Kolmogorov complexity, tailored to our needs. For a complete treatment of this subject, see [17] or the survey [16].

Fix a universal Turing machine U with input alphabet Σ. The machine U takes two parameters p and y. U interprets p as a program and simulates p on input y. The Kolmogorov complexity of a string $x \in \Sigma^*$, given $y \in \Sigma^*$, is defined as

$$K_U(x|y) = \min\{|p| : U(p, y) = x\}.$$

Because one can prove an invariance theorem that claims that Kolmogorov complexity, defined with respect to any two different universal machines, differs by only an additive constant, we will drop the subscript U. In fact, for the purpose of our analysis, one fixed universal Turing machine is always assumed. Thus $K(x|y)$ is the *minimum* number of *digits* (i.e., letters in Σ) in a description from which x can be effectively reconstructed, given y. Let $K(x) = K(x|\lambda)$, where λ denotes the null string.

By simple counting, for each n, $c < n$, and y, there are at least $|\Sigma|^n - |\Sigma|^{n-c} + 1$ distinct x's of length n with the property

$$K(x|y) \geq n - c. \tag{1}$$

We call a string x of length n *random* if

$$K(x) \geq n - \log n,$$

where the logarithm is taken over base $|\Sigma|$. Sometimes, we need to encode x in a *self-delimiting* form $\bar{x}$ to be able to decompose $\bar{x}y$ into x and y. One possible coding for $\bar{x}$ may be $10L(x)10x$, where $L(x)$ is the binary representation of $|x|$ with each bit doubled. (Assume that $0, 1 \in \Sigma$.) For example, if $x = 0000000$, $|x| = 111$ in binary, and $L(x) = 111111$. This costs us about $2 \log |x|$ extra bits. Thus, the self-delimiting representation $\bar{x}$ of x requires at most $|x| + 2 \log |x| + 4$ digits [16]. Note that our Kolmogorov complexity is a bit unconventional, since here we consider strings over the arbitrary fixed alphabet Σ instead of binary strings.

4.2. Longest common subsequences: The average case. We have shown that the LCS problem cannot be approximated with ratio n^δ for some $\delta > 0$ in polynomial time unless **NP** = **P**. Thus no polynomial-time algorithm can produce even approximately *long* common subsequences. However, this claim only holds for the (probably extremely rare) worst cases. Here we would like to show that for random input sequences, the LCS problem can be approximated up to a small additive error.

THEOREM 4.1. *For an input set S containing n sequences of length n, the algorithm Long-Run approximates the LCS with an expected additive error $O(n^{1/2+\epsilon})$ for arbitrarily small $\epsilon > 0$.*

The proof is based on the following two lemmas which give a lower bound on the performance of Long-Run and an upper bound on the length of an LCS for a set of Kolmogorov random strings.

LEMMA 4.2. *Let $\epsilon > 0$ be any constant and x, a string of length n. If some letter $a \in \Sigma$ appears in x less than $\frac{n}{k} - n^{1/2+\epsilon}$ times or more than $\frac{n}{k} + n^{1/2+\epsilon}$ times, then for some constant $\delta > 0$,*

$$K(x_i) \leq n - \delta n^{2\epsilon}.$$

Proof. In principle, this result can be proved using methods in [18]. By encoding each letter as a binary string of $\log_2 k$ bits, the results in [18] imply that, when $\log_2 k$ is an integer, the lemma is true. To show the general case, we simply do a direct estimation as follows.

Suppose that for some letter $a \in \Sigma$, x contains only $d = \frac{n}{k} - n^{1/2+\epsilon}$ a's. There are only

$$\binom{n}{d} (k-1)^{n-d}$$

strings of length n with d occurrences of a. Taking a logarithm with base k would give us the number of digits specifying x_i. By an elementary estimation (using Stirling's formula), we can show that

$$\log_k \binom{n}{d} (k-1)^{n-d} \leq n - \delta n^{2\epsilon} \tag{2}$$

for some $\delta > 0$. Thus $K(x_i) \leq n - \delta n^{2\epsilon}$. □

Now consider a fixed Kolmogorov random string x of length n^2. Cut x into n equal-length pieces $x_1, \ldots, x_n$. It follows from Lemma 4.2 and the randomness of x that, for each $a \in \Sigma$, a appears in each of $x_1, \ldots, x_n$ at least $\frac{n}{k} - O(n^{1/2+\epsilon})$ times for any small $\epsilon > 0$. Thus

$$a^{n/k - O(n^{1/2+\epsilon})} \tag{3}$$

is a common subsequence of sequences $x_1, \ldots, x_n$. In the next lemma, we show that an LCS for $x_1, \ldots, x_n$ cannot really be much longer than the one in formula (3).

LEMMA 4.3. *For any common subsequence s of $x_1, \ldots, x_n$,*

$$|s| < \frac{n}{k} + n^{\frac{1}{2}+\epsilon}. \tag{4}$$

Proof. For the purpose of making a contradiction, suppose that

$$|s| \geq \frac{n}{k} + n^{\frac{1}{2}+\epsilon}.$$

We will try to compress x using s. The idea is to save n^δ digits for some $\delta > 0$ on each x_i.

Fix an x_i. We do another encoding of x_i. Let $s = s_1 s_2 \ldots s_p$, $p = |s|$. We align the letters in s with the corresponding letters in x_i greedily from left to right and rewrite x_i as follows:

$$\alpha_1 s_1 \alpha_2 s_2 \ldots \alpha_p s_p x_i'. \tag{5}$$

Here α_1 is the longest prefix of x_i containing no s_1, α_2 is the longest substring of x_i starting from s_1 containing no s_2, and so on. x_i' is the remaining part of x_i after s_n. Thus α_j does not contain letter s_j for $j = 1, \ldots, p$. That is, each α_j contains at most $k-1$ letters in Σ. In other words, $\alpha_j \in (\Sigma - \{s_j\})^*$.

We now show that x_i can be compressed by at least n^δ digits for some $\delta > 0$ with the help of s. In fact, it is sufficient to prove that the prefix

$$y = \alpha_1 s_1 \alpha_2 s_2 \ldots \alpha_p s_p, \tag{6}$$

can be compressed by this amount.

Using s, we know which $k-1$ letters in Σ appear in α_i for each i. Thus we can recode y as follows. For each i, if $s_i = a_k$, then do nothing. Otherwise, replace s_i by a_k, the last letter in Σ, and in α_i, replace every a_k by s_i. We can convert this transformed string y' back to y using s by reversing above process.

Now this transformed string y' is also a string over Σ. But in y', letter a_k appears $\frac{n}{k} + n^{1/2+\epsilon}$ times, since $|s|$ is at least this long. By Lemma 4.2, for some constant $\delta > 0$, we have

$$K(y') \leq |y'| - \frac{\delta n^{2\epsilon}}{k}.$$

But from y' and s we can obtain y, and hence, together with a self-delimiting form of x_i', we can obtain x_i. We conclude that

$$K(x_i|s) \leq n - \frac{\delta n^{2\epsilon}}{k} + O(\log n),$$

where the term $O(\log n)$ takes care of the extra digits required for the self-delimiting representation of x_i' and the description of the above procedure.

We repeat the above argument for every x_i. In total, we save $\Omega(n^{1+2\epsilon})$ digits encoding x. Thus,

$$K(x) \leq n^2 - \Omega(n^{1+2\epsilon}) + |s| + O(n \log n) < |x| - \log |x|.$$

Therefore, x is not random, and we have a contradiction! □

We are now ready to prove the theorem.

Proof of Theorem 4.1. Consider all possible inputs of n sequences of length n. For each such input, we concatenate the n sequences together to obtain one string. Only about $1/n^2$ of them are not random by formula (1). That is, only $1/n^2$ of them do not satisfy $K(x) \geq |x| - \log |x|$. For all the others, the above lower and upper bounds apply, and the algorithm Long-Run produces a common subsequence that is at most $O(n^{1/2+\epsilon})$ shorter than the LCS for any fixed $\epsilon > 0$. Observe that the worst-case error of Long-Run is $(k-1)n/k$. Thus, a simple averaging shows that the expected error of Long-Run is $O(n^{1/2+\epsilon})$ for any fixed $\epsilon > 0$.

Lemmas 4.2 and 4.3 actually give very tight upper and lower bounds on the expected LCS length of n random sequences of length n. We note in passing that the same problem for two sequences is still open and there is a large gap between the current best upper and lower bounds [2], [5], [24].

COROLLARY 4.4. *The expected length of an LCS for a set of n random sequences of length n is* $\frac{n}{k} \pm n^{1/2+\epsilon}$ *for any* $\epsilon > 0$.

4.3. Shortest common supersequences: The average case. In [8], the performance of the following algorithm for the SCS problem is analyzed.

ALGORITHM MAJORITY-MERGE

1. Input: n sequences, each of length n.
2. Set supersequence $s :=$ *null string*;
3. Let a be the majority among the leftmost letters of the remaining sequences. Set $s := sa$ and delete the front a from these sequences. Repeat this step until no sequences are left.
4. Output s.

It is shown in [8] that, on an alphabet of size k, Majority-Merge produces a common supersequence of length $O(n \log n)$ in the worst case and a common supersequence of length $(k+1)n/2 + O(\sqrt{n})$ on the average. In the next theorem, we will show that its average-case performance is actually near optimal.

THEOREM 4.5. *For a set S containing n sequences over* Σ *of length n, the algorithm Majority-Merge approximates the SCS with an expected additive error* $O(n^{\delta})$, *where* $\delta = \sqrt{2}/2 \approx 0.707$.

As in the proof of Theorem 4.1, we first consider Kolmogorov random strings and obtain some tight upper bound on the performance of the algorithm Majority-Merge and a lower bound on the length of the SCS. So, again, fix a Kolmogorov random string x of length n^2 and cut x into n equal-length pieces $x_1, \ldots, x_n$. Let $S = \{x_1, \ldots, x_n\}$. It is shown in [8] that Majority-Merge actually finds a common supersequence of length $(k+1)n/2 + O(\sqrt{n})$ on

the set S. Hence, it suffices to prove that $\mathrm{OPT}(S) \geq (k+1)n/2 - O(n^\delta)$. We prove this in what follows by using essential properties of Kolmogorov random strings.

Fix an arbitrary SCS $s = s_1 s_2 \ldots s_l$ for S and let $\mathcal{A}$ denote the procedure that produces s on set S by scanning the input sequences from left to right and merging common letters. We want to show that

$$l \geq \frac{(k+1)n}{2} - O(n^\delta).$$

Note that (i) clearly $l \leq kn$ and (ii) $\mathcal{A}$ is uniquely determined by the SCS s. Let us arrange the input sequences $x_1, \ldots, x_n$ as an $n \times n$ matrix M with x_i as row i. For each $1 \leq i \leq l$, call the sequence from top to bottom, consisting of the first letters in the remaining input sequences after $i-1$ steps of $\mathcal{A}$, the ith *frontier*. Thus a frontier is just a jagged line from top to bottom, indicating which letter is being considered by $\mathcal{A}$ at the moment in each sequence x_i.

Since $\mathcal{A}$ totally merges n^2 letters in producing s, the number n^2/l represents the *average number* of letters merged by $\mathcal{A}$ in one step. We want to show that on the average, $\mathcal{A}$ merges at most $2n/(k+1) + O(n^\delta)$ letters in a step for some $\delta < 1$, i.e.,

$$\frac{n^2}{l} \leq \frac{2n}{(k+1)} + O(n^\delta).$$

This would imply that $l \geq (k+1)n/2 - O(n^\delta)$.

CLAIM 4.6. *On the average, $\mathcal{A}$ merges at most $2n/(k+1) + O(n^\delta)$ letters in a step.*

Proof. The basic idea is to show that the average merge amount takes its maximum when the supersequence s is of the form π^* for some permutation π of alphabet Σ, using the property that after each merge, the successors of the merged letters are "generated" according to a fair-coin rule, i.e., the letters $a_1, \ldots, a_k$ must be distributed evenly among these "new" letters. This property holds because the matrix M is random.

For each $1 \leq i \leq l$ and $1 \leq j \leq k$, let $r_{i,j}$ denote the number of letter a_j contained in frontier i. Define $r_i = \sum_{j=1}^{k} r_{i,j}$ for each i as the *length* of frontier i. Clearly, $n = r_1 \geq \cdots \geq r_l$. Let l_0 be the smallest index such that $r_{l_0} \leq 2n/(k+1)$. Then we only need to prove an upper bound of $2n/(k+1) + O(n^\delta)$ on the average amount of merges made by $\mathcal{A}$ up to step l_0, since it merges at most $2n/(k+1)$ letters every step after step l_0. For each $1 \leq i \leq l_0$, denote the number of letters merged at step i as m_i.

First we would like to show that, at any step, if a large number of some letter a is merged, then the *new* letters immediately behind these a's in the involved input sequences should have approximately equal share of the letters $a_1, \ldots, a_k$. In view of Lemma 4.2, this is true as long as we can show that the subsequence consisting of these new letters in the next frontier is more or less random. We need the following lemma, which indirectly proves the randomness of this subsequence.

LEMMA 4.7. *Let* frontier(i) *be the list of positions indicating where the ith frontier cuts through sequences $x_1, \ldots, x_n$. Let M' be the all the letters on the left of the ith frontier including the ith frontier, plainly listed column by column from left to right. Then,*

$$K(\text{frontier}(i) | s, i, M', n) \leq O(1).$$

Proof. Given s, i, M', n, we can simulate $\mathcal{A}$, together with s, on partial input M' for $i-1$ steps. Then we should have the positions for the ith frontier. Note that in the listing of M', it is not necessary that each column is of length n, since some input sequences may have already run out. But this can be detected easily using n. Thus in all the future steps, we know that these sequences are not present any more and can correctly arrange the letters of M'. □

In the following calculation, let ϵ be the solution of the following equation:

$$1 - 2\epsilon\delta = \delta,$$

where $\delta = \frac{1}{2} + \epsilon$. So $\epsilon = (\sqrt{2} - 1)/2 \approx 0.207$ and $\delta = \sqrt{2}/2 \approx 0.707$.

Let $i < l_0$ be any index. Suppose that the letter $a_j \in \Sigma$ is chosen to be merged by $\mathcal{A}$ at step i. If the letter a_j appears in the ith frontier $\Omega(n^\delta)$ times, i.e., $r_{i,j} = \Omega(n^\delta)$, then by Lemma 4.2, if the letters $a_1, \ldots, a_k$ are unevenly distributed among the subsequence of frontier $i + 1$ consisting of the successors of these merged a_j's, then we can compress this subsequence by $\Omega(n^{2\epsilon\delta})$ letters.

LEMMA 4.8. *There are at most $O(n^{1-2\epsilon\delta}) = O(n^\delta)$ subsequences of length $n^d = \Omega(n^\delta)$ such that some letter appears $n^{d/2+\epsilon}$ more often (or less often) than the average n^d/k in these subsequences.*

Proof. Otherwise we can describe the random matrix M by simply listing all the letters other than those appearing in the subsequences mentioned above, recording the supersequence s and the locations (i.e., indices) of the subsequences and compressing the subsequences using s. Since we save $\Omega(n^{2\epsilon\delta})$ letters on each such subsequence by Lemma 4.2, in total we save more than $\Omega(n^{1-2\epsilon\delta}) \cdot \Omega(n^{2\epsilon\delta}) = \Omega(n)$ letters. Thus we can encode x in less than $|x| - \log|x|$ letters. Note that by Lemma 4.7, the position of the letters of frontier i in their corresponding input sequence can be derived from s, i, n, and the preceding frontiers. □

Let the frontiers containing the $q = O(n^\delta)$ unevenly distributed subsequences be indexed $p_1, \ldots, p_q$. Let $p_0 = 1$ and $p_{q+1} = l_0 + 1$. Cut M into $q + 1$ sections $M_0, \ldots, M_q$, where M_i begins at frontier p_i and ends at frontier $p_{i+1} - 1$.

Now we fix a section M_g and calculate the total amount of merges made by $\mathcal{A}$ in M_g. Since the letters in each set of new letters (after a merge) are distributed uniformly within this section, we have the following relation between $r_{i+1,j}$ and $r_{i,j}$. Suppose that the letter a_h is merged at step i, $p_g \le i < p_{g+1}$. Then $r_{i+1,j} = r_{i,j} + r_{i,h}/k \pm O(n^\delta)$ for each $j \ne h$, and $r_{i+1,h} = r_{i,h}/k \pm O(n^\delta)$. (Note that the recurrence is automatically true if $r_{i,h} < O(n^\delta)$.) To simplify the presentation, we will drop the minor term $O(n^\delta)$ below when using the above recurrence relation and simply add $O(n^\delta) \cdot (p_{g+1} - p_g)$ later to the total amount of merges in section M_g. It is easy to verify that in the worst case this fluctuation of magnitude $O(n^\delta)$ can add at most $k \cdot O(n^\delta) = O(n^\delta)$ to the average merge amount.

Define a function $\rho(t_1, \ldots, t_k, i, j)$ satisfying the following recurrence relation:

$$\begin{aligned}
\rho(t_1, \ldots, t_k, i, 1) &= t_i, \quad 1 \le i \le k, \\
\rho(t_1, \ldots, t_k, j \bmod k, j+1) &= \frac{\rho(t_1, \ldots, t_k, j \bmod k, j)}{k}, \\
\rho(t_1, \ldots, t_k, j+1 \bmod k, j+1) &= \rho(t_1, \ldots, t_k, j+1 \bmod k, j) \\
&\quad + \frac{\rho(t_1, \ldots, t_k, j \bmod k, j)}{k}, \\
&\vdots \\
\rho(t_1, \ldots, t_k, j+k-1 \bmod k, j+1) &= \rho(t_1, \ldots, t_k, j+k-1 \bmod k, j) \\
&\quad + \frac{\rho(t_1, \ldots, t_k, j \bmod k, j)}{k}.
\end{aligned}$$

Intuitively, the function $\rho(t_1, \ldots, t_k, i, j)$ corresponds to the distribution of letters $a_1, \ldots, a_k$ in the frontiers if the letters are merged following the sequence π^*, where $\pi = a_{i_1} \ldots a_{i_k}$ is

some permutation of Σ and, initially, the letters $a_{i_1}, \ldots, a_{i_k}$ are distributed as $(t_1, \ldots, t_k)$. Let

$$\mu(t_1, \ldots, t_k, i) = \sum_{j=1}^{i} \rho(t_1, \ldots, t_k, j \bmod k, j).$$

Thus $\mu(t_1, \ldots, t_k, i)$ represents the total amount of merges achieved following the sequence π^* for i steps, given the initial distribution $(t_1, \ldots, t_k)$ of letters $a_{i_1}, \ldots, a_{i_k}$. It can be shown that

$$\mu(t_1, \ldots, t_k, i) = \sum_{j=1}^{k} \omega(i - j + 1) \cdot t_j,$$

where $\omega(i)$ is a function defined recursively as follows:

$$\omega(i) = 0, \quad i \le 0,$$

$$\omega(i) = 1 + \sum_{j=1}^{k} \omega(i - j)/k, \quad i \ge 1.$$

Intuitively, the number $\omega(i - j + 1)$ represents the total contribution to the amount of merges achieved in i steps from a letter that is merged at step j, $1 \le j \le i$. Note that $\omega(i)$ is an increasing function. We can prove the following lemma by induction.

LEMMA 4.9. *Let i be any index such that $p_g \le i \le p_{g+1} - 1$. Suppose that $r_{i,j_1} \ge \cdots \ge r_{i,j_k}$, where $j_1, \ldots, j_k$ is some permutation of $1, \ldots, k$. Then $\sum_{j=i}^{p_{g+1}-1} m_j \le \mu(r_{i,j_1}, \ldots, r_{i,j_k}, p_{g+1} - i)$.*

Proof. The lemma holds clearly if $i = p_{g+1} - 1$. Now we prove that the lemma holds for i, assuming that it holds for $i + 1$. For convenience, here we assume that $j_1 = 1, \ldots, j_k = k$. Suppose that $\mathcal{A}$ merges the letter a_h at step i. Then

$$\begin{aligned}\sum_{j=i}^{p_{g+1}-1} m_j &= r_{i,h} + \sum_{j=i+1}^{p_{g+1}-1} m_j \\ &\le r_{i,h} + \mu(r_{i+1,1}, \ldots, r_{i+1,h-1}, r_{i+1,h+1}, \ldots, r_{i+1,k}, r_{i+1,h}, p_{g+1} - i - 1),\end{aligned}$$

where $r_{i+1,j} = r_{i,j} + r_{i,h}/k$ for each $j \ne h$, and $r_{i+1,h} = r_{i,h}/k$. Observe that $r_{i+1,1} \ge \cdots \ge r_{i+1,h-1} \ge r_{i+1,h+1} \ge \cdots \ge r_{i+1,k} \ge r_{i+1,h}$. However,

$$\begin{aligned}&r_{i,h} + \mu(r_{i+1,1}, \ldots, r_{i+1,h-1}, r_{i+1,h+1}, \ldots, r_{i+1,k}, r_{i+1,h}, p_{g+1} - i - 1) \\ &= \mu(r_{i,h}, r_{i,1}, \ldots, r_{i,h-1}, r_{i,h+1}, \ldots, r_{i,k}, p_{g+1} - i) \\ &= \omega(p_{g+1} - i) \cdot r_{i,h} + \omega(p_{g+1} - i - 1) \cdot r_{i,1} + \cdots + \omega(p_{g+1} - i - h + 1) \cdot r_{i,h-1} \\ &\quad + \omega(p_{g+1} - i - h) \cdot r_{i,h+1} + \cdots + \omega(p_{g+1} - i - k + 1) \cdot r_{i,k} \\ &\le \omega(p_{g+1} - i) \cdot r_{i,1} + \cdots + \omega(p_{g+1} - i - k + 1) \cdot r_{i,k} \\ &= \mu(r_{i,1}, \ldots, r_{i,k}, p_{g+1} - i).\end{aligned}$$

The above inequality holds because $\omega(j)$ is increasing and $r_{i,1} \ge \cdots \ge r_{i,k}$. □

Hence, the total amount of merges made by $\mathcal{A}$ in section M_g is

$$\begin{aligned}\sum_{j=p_g}^{p_{g+1}-1} m_j &\le \mu(r_{p_g,1}, \ldots, r_{p_g,k}, p_{g+1} - p_g)\\ &= \sum_{j=1}^{k} \omega(p_{g+1} - p_g + 1 - j) \cdot r_{p_g,j}\\ &\le \sum_{j=1}^{k} \omega(p_{g+1} - p_g) \cdot r_{p_g,j}\\ &= \omega(p_{g+1} - p_g) \cdot \sum_{j=1}^{k} r_{p_g,j}\\ &\le \omega(p_{g+1} - p_g) \cdot n.\end{aligned}$$

We need one more lemma.

LEMMA 4.10. $\omega(i) \le \frac{2i}{k+1} + 1$.

Proof. Clearly, the lemma holds for all $i \le 1$. Now suppose that it holds for all $i \le h$ for some h. Then

$$\begin{aligned}\omega(h+1) &= 1 + \sum_{j=1}^{k} \omega(h+1-j)/k\\ &\le 2 + \sum_{j=1}^{k} \frac{2(h+1-j)}{k(k+1)}\\ &= 2 + \frac{2(h+1) - k - 1}{k+1}\\ &= 1 + \frac{2(h+1)}{k+1}. \qquad \square\end{aligned}$$

Therefore,

$$\sum_{j=p_g}^{p_{g+1}-1} m_j \le \omega(p_{g+1} - p_g) \cdot n \le \frac{2n(p_{g+1} - p_g)}{k+1} + n.$$

Now we add $O(n^\delta) \cdot (p_{g+1} - p_g)$ back to the above bound and conclude that the total amount of merges in section M_g is at most

$$\frac{2n(p_{g+1} - p_g)}{k+1} + n + O(n^\delta) \cdot (p_{g+1} - p_g).$$

Thus, the total amount of merges in all sections is

$$\begin{aligned}&\sum_{g=0}^{q} \frac{2n(p_{g+1} - p_g)}{k+1} + n + O(n^\delta) \cdot (p_{g+1} - p_g)\\ &\quad= \frac{2nl_0}{k+1} + n(q+1) + O(n^\delta) \cdot l_0\\ &\quad= \frac{2nl_0}{k+1} + n \cdot O(n^\delta) + O(n^\delta) \cdot l_0\\ &\quad= \frac{2nl_0}{k+1} + O(n^\delta) \cdot l_0.\end{aligned}$$

That is, the overall average amount of merges is $\frac{2n}{k+1} + O(n^{\delta}) = \frac{2n}{k+1} + O(n^{0.707})$. This completes the proof of the claim. □

Proof of Theorem 4.5. As in the argument at the end of the proof of Theorem 4.1, and because Majority-Merge produces a common supersequence of length $O(n \log n)$ in the worst case [8], the algorithm has an expected additive error $O(n^{0.707})$.

The above proof implies the following interesting corollary.

COROLLARY 4.11. *The expected length of an SCS for a set of n random sequences of length n is* $(k+1)n/2 \pm O(n^{0.707})$.

As we mentioned before, both Theorems 4.1 and 4.5 actually hold for inputs consisting of $p(n)$ sequences of length n, where $p(\,)$ is some fixed polynomial.

5. Some remarks. A problem that is closely related to the SCS problem is the shortest common superstring problem. Although this problem is also **NP**-hard, the status of its approximation complexity is quite different. Blum et al. have shown that shortest common superstring problem can be approximated within a factor of 3 in polynomial time [4]. They also showed that the problem (on an unbounded alphabet) is **MAX SNP**-hard.

Acknowledgments. We thank the referees for many constructive criticisms and suggestions, and we thank C. Fraser for a correction.

REFERENCES

[1] A. AHO, J. HOPCROFT, AND J. ULLMAN, *Data Structures and Algorithms*, Addison–Wesley, Reading, MA, 1983.

[2] K. ALEXANDER, *The rate of convergence of the mean length of the longest common subsequence*, 1992, manuscript.

[3] A. ARORA, C. LUND, R. MOTWANI, M. SUDAN, AND M. SZEGEDY, *Proof verification and hardness of approximation problems*, in Proc. 33rd IEEE Symposium on Foundations of Computer Science, Pittsburgh, PA, 1992, pp. 14–23.

[4] A. BLUM, T. JIANG, M. LI, J. TROMP, AND M. YANNAKAKIS, *Linear approximation of shortest superstrings*, in Proc. 23rd ACM Symposium on Theory of Computing, 1991, pp. 328–336; J. Assoc. Comput. Mach., to appear.

[5] V. CHVATAL AND D. SANKOFF, *Longest common subsequences of two random sequences*, J. Appl. Probab. 12 (1975), pp. 306–315.

[6] M. O. DAYHOFF, *Computer analysis of protein evolution*, Sci. Amer., 221 (1969), pp. 86–95.

[7] D. E. FOULSER, *On random strings and sequence comparisons*, Ph.D. thesis, Computer Science Department, Stanford University, 1986.

[8] D. E. FOULSER, M. LI, AND Q. YANG, *Theory and algorithms for plan merging*, Artificial Intelligence, 57 (1992), pp. 143–181.

[9] GAREY AND D. JOHNSON, *Computers and Intractability*, W. H. Freeman, New York, 1979.

[10] C. C. HAYES, *A model of planning for plan efficiency: Taking advantage of operator overlap*, in Proc. 11th International Joint Conference of Artificial Intelligence, Detroit, Michigan, 1989, pp. 949–953.

[11] D. S. HIRSCHBERG, *The longest common subsequence problem*, Ph.D. thesis, Princeton University, 1975.

[12] W. J. HSU AND M. W. DU, *Computing a longest common subsequence for a set of strings*, BIT 24 (1984), pp. 45–59.

[13] R. W. IRVING AND C. B. FRASER, *Two algorithms for the longest common subsequence of three (or more) strings*, in Proc. Symposium on Combinatorial Pattern Matching, Tucson, AZ, 1992.

[14] D. KARGER, R. MOTWANI, AND G. D. S. RAMKUMAR, *On approximating the longest path in a graph*, in Proc. Workshop on Algorithms and Data Structure, Montreal, Canada, 1993, pp. 421–432.

[15] R. KARINTHI, D. S. NAU, AND Q. YANG, *Handling feature interactions in process planning*, J. Appl. Artif. Intell., 6 (1992), pp. 389–415.

[16] M. LI AND P. M. B. VITÁNYI, *Kolmogorov complexity and its applications*, in Handbook of Theoretical Computer Science, Vol. A, J. van Leeuwen, ed., Elsevier/MIT Press, New York, NY, Cambridge, MA, 1990, pp. 187–254.

[17] ———, *An Introduction to Kolmogorov Complexity and Its Applications*, Springer-Verlag, New York, Berlin, Heidelberg, 1993.

[18] M. LI AND P. M. B. VITÁNYI, *Combinatorial properties of finite sequences with high Kolmogorov complexity*, Math. Systems Theory, to appear.

[19] S. Y. LU AND K. S. FU, *A sentence-to-sentence clustering procedure for pattern analysis*, IEEE Trans. Systems., Man Cybernet, SMC-8(5) (1978), pp. 381–389.

[20] C. LUND AND M. YANNAKAKIS, *On the hardness of approximating minimization problems*, in Proc. ACM Symposium Theory of Computing, San Diego, CA, 1993, pp. 286–293.

[21] D. MAIER, *The complexity of some problems on subsequences and supersequences* J. Assoc. Comput. Mach., 25 (1978), pp. 322–336.

[22] C. H. PAPADIMITRIOU AND M. YANNAKAKIS, *Optimization, approximation, and complexity classes*, J. Comput. System Sci., 43 (1991), pp. 425–440.

[23] K. RAIHA AND E. UKKONEN, *The shortest common supersequence problem over binary alphabet is NP-complete*, Theoret. Comput. Sci., 16 (1981), pp. 187–198.

[24] D. SANKOFF AND J. KRUSKALL, EDS., *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison–Wesley, Reading, MA, 1983.

[25] T. SELLIS, *Multiple query optimization*, ACM Trans. Database Systems, 13 (1988), pp. 23–52.

[26] T. F. SMITH AND M. S. WATERMAN, *Identification of common molecular subsequences*, J. Molecular Biology, 147 (1981), pp. 195–197.

[27] J. STORER, *Data Compression: Methods and Theory*, Computer Science Press, Rockville, MD, 1988.

[28] V. G. TIMKOVSKII, *Complexity of common subsequence and supersequence problems and related problems* Kibernetika, 5 (1989), pp. 1–13. (English translation.)

[29] R. A. WAGNER AND M. J. FISCHER, *The string-to-string correction problem* J. Assoc. Comput. Mach., 21 (1974), pp. 168–173.

SIAM J. COMPUT.
Vol. 24, No. 6, pp. 1141–1156, December 1995

ON THE GENERATION OF RANDOM BINARY SEARCH TREES*

LUC DEVROYE† AND JOHN MICHAEL ROBSON‡

Abstract. We consider the computer generation of random binary search trees with n nodes for the standard random permutation model. The algorithms discussed here output the number of external nodes at each level, but not the shape of the tree. This is important, for example, when one wishes to simulate the height of the binary search tree. Various paradigms are proposed, including depth-first search with pruning, incremental methods in which the tree grows with random-sized jumps, and a tree growing procedure gleaned from birth-and-death processes. The last method takes $O(\log^4 n)$ expected time.

Key words. binary search tree, height of a tree, probabilistic analysis, expected complexity, simulation, random combinatorial object, point process, recursive procedure

AMS subject classification. 68Q25, 68U20, 93E30, 00A72, 11K45, 65C05, 65C10, 05C80, 68R++, 68P05, 68P10, 60C05

1. Introduction.

The binary search tree is one of the most frequently used structures in computer science, see e.g., Aho, Hopcroft, and Ullman [1], Knuth [19], or Cormen, Leiserson, and Rivest [4]. A random binary search tree is defined as the random binary tree obtained by consecutive insertion of $X_1, \ldots, X_n$ into an initially empty tree, where $X_1, \ldots, X_n$ is either an independently and identically distributed (i.i.d.) sequence of random variables with a fixed density, or an equiprobable random permutation of $\{1, \ldots, n\}$. The height H_n of the tree is the maximal distance between any node and the root (thus, $H_1 = 0$, as the root is at distance 0 from itself, and $H_2 = 1$). In this paper, we propose various methods for the generation of random binary search trees. Trees with n nodes necessarily require $\Omega(n)$ time. Since many quantities related to random trees such as their heights grow logarithmically with n, large size trees are required in simulations that attempt to extract asymptotic information. In fact, one often does not care about the actual tree, but rather about the number of nodes at each level. The methods given below take sublinear expected time and output a random vector $(n_0, \ldots, n_m)$ where n_i is the number of external nodes at level i. The height H_n is nothing but $m - 1$, for example. Early studies of H_n include Robson [26], [27], Pittel [23], and Mahmoud and Pittel [21]. See also Mahmoud [20]. While it is known that

$$\frac{H_n}{\log n} \to c \stackrel{\text{def}}{=} 4.33107\ldots \text{a.s.}$$

as $n \to \infty$ (Devroye, [6], [9]), very little additional information is available regarding H_n, and one is led to simulation in order to study the second-order properties of H_n such as its variance, its asymptotic distribution, and so forth. Such simulations require formidable values of n in view of the logarithmic growth of H_n with n. It is thus of great importance to ensure that the time and especially the space requirements grow slowly with n.

Constructing the tree by consecutive insertions leads to a $\Theta(n \log n)$ expected time and $\Theta(n)$ space algorithm. One can exploit a certain growth property of the random binary search tree (all external nodes are equally likely to receive a new node), leading to a $\Theta(n)$ time and $\Theta(n)$ space method for growing the tree by direct insertion of new nodes at old leaves. With extra effort, the space requirement can be reduced to $\Theta(\log n)$ expected space.

* Received by the editors January 22, 1992; accepted for publication (in revised form) March 15, 1994.

†School of Computer Science, McGill University, 3480 University Street, Montreal, Quebec H3A 2K6, Canada (luc@crodo.cs.mcgill.ca). This author's research was sponsored by National Science and Engineering Research Council of Canada grant A3456 and Formation de chercheurs et action de recherche grant 90-ER-0291.

‡Department of Computer Science, Australian National University, G.P.O. Box 4, Canberra, A.C.T. 2601, Australia. Current address: LaBRI, Université Bordeaux I, 351 Cours de la Libiration, 33405 Talence cedex, France (robson@essi2.cerisi.tr).

In §3, we present a simple method that requires $O(n/\log n)$ expected time and $O(\log n)$ expected space. A shortcut is required as a sublinear time method cannot possibly consider all nodes in the tree. The algorithm presented here uses some sort of depth-first search with tree pruning gleaned from game tree search applications. The idea of simulating random variables via shortcuts that bypass the definition of the random variables has been exploited by many. For example, the maximum of n i.i.d. random variables with a given density f can be generated in expected time $O(1)$ or $O(\log n)$ depending upon whether the distribution function is available at unit cost or not [28], [5]. The sum of n i.i.d. random variables can also be generated in time $o(n)$; often, $O(1)$ expected time is achievable, as is demonstrated in [7], [11]. A binomial (n, p) random variable representing the outcome of n coin flips is nowadays routinely generated in $O(1)$ expected time ([2], [15], [24], [25], [10], [16], [29]-[30]) uniformly bounded over n and p. The result of this paper is just another illustration of the same principle. The method of §3 is extremely easy to implement, and is competitive with the other methods for values n that are not too large.

In §4, we recall a simple linear time method based upon growing a random binary search tree by replacing external nodes, i.e., not by insertion from the root down. Probabilistic shortcuts based upon waiting times allow us in §5 to reduce the expected time to $O(\sqrt{n}\log n)$. This requires efficient generators for a multivariate hypergeometric and a certain waiting time distribution, thereby rendering the programming effort and the overhead a bit heavier. Nevertheless, for medium-sized values of n, the method is very useful.

In §6, a random binary search tree is grown by imagining that each external node is a living organism that will bear two children and die according to a simple Poisson point process. We then let the time grow by constant amounts, so that the tree grows at an exponential rate. At any given moment, we have a correctly distributed binary search tree, but the size is random. When one stops as soon as the size of the tree is n or larger, the expected time complexity is $O(\log^4 n)$. A modification of the algorithm is introduced to obtain the right size. The constant in $O(\log^4 n)$ is rather large due to the overhead in a multinomial random vector generator used in a bottleneck portion of the algorithm. This last method is useful for extremely large n, such as $n \approx 10^{40}$.

2. Two key properties of random binary search trees.

FACT 1. *If we associate with node i in a random binary search tree an integer S_i denoting the size of the subtree rooted at that node, then for any (i.e., left or right) child j of node i, we have*

$$S_j \stackrel{\mathcal{L}}{=} \lfloor S_i U \rfloor ,$$

where $\stackrel{\mathcal{L}}{=}$ means "is distributed as," and U is a uniform $[0, 1]$ random variable independent of S_i.

As an immediate corollary of this we have the following fact.

FACT 2. *Let node i in a random binary search tree have children j and k. Then*

$$\max(S_j, S_k) \stackrel{\mathcal{L}}{=} \lfloor S_i \max(U, 1-U) \rfloor , \stackrel{\mathcal{L}}{=} \left\lfloor S_i \frac{1+U}{2} \right\rfloor ,$$

where U is a uniform $[0, 1]$ random variable independent of S_i.

3. A simple algorithm for the height of a random binary search tree.

3.1. Description. The algorithm keeps a stack of nodes characterized by a pair (l, s), where l is the level of a node (i.e., its distance from the root), and s is the size of the subtree rooted at that node. The actual position of each node in the tree and the actual tree with all its links are never constructed. As we proceed, nodes on the stack are split into two nodes representing the children. A node (l, s) is thus split into $(l + 1, s_1)$ and $(l + 1, s_2)$, where $s_1 + s_2 = s - 1$. Moreover, s_1 is distributed as $\lfloor sU \rfloor$, where U is uniformly distributed on [0, 1]. See Fact 1 above. The new nodes are put back on the stack with the largest subtree on top. A node (l, s) can at best produce a node at distance $l + s - 1$ from the root, so there is no point in processing or stacking such a node if $l + s - 1$ does not exceed the current value of the height of the tree, i.e., the largest level among all nodes seen thus far. This observation is at the basis of the sublinear expected time: not all nodes in the random binary search tree are expanded; in fact, only a negligible fraction ($O(1/\log n)$) is ever expanded.

```
A sublinear depth-first search algorithm.

makenull (S) (S is a stack).
push ((0, n), S) (put the root onto the stack).
h ← 0 (h is the current value of the height).
while not empty (S) do
  pop ((l, s), S)
  if l + s − 1 > h then
    l ← l + 1
    if l > h then h ← l
    generate U uniformly on [1/2, 1].
    s2 ← ⌊sU⌋, s1 ← s − 1 − s2.
    if s1 > 1 then if l + s1 − 1 > h then push ((l, s1), S).
    if s2 > 1 then if l + s2 − 1 > h then push ((l, s2), S).
return h.
```

3.2. Expected stack size. It is easy to verify by induction that every level except the furthest can occur at most once on the stack; the furthest occurs at most twice (just note that the level numbers of the nodes on the stack are strictly increasing except possibly for the top node). This shows without further ado that the maximal stack size is less than one plus the height of the tree, and this in turn is less than $(c + \epsilon) \log n$ almost surely for any $\epsilon > 0$. Furthermore, the expected stack size is not greater than $(c + o(1)) \log n$.

3.3. Expected time analysis. Initially, we push and pop a sequence of nodes that correspond to a path down the tree ending with a node with a subtree of size one. After that, the algorithm backtracks by processing a node closer to the root. Let us call L_n the level of this furthest node encountered in the first phase of the algorithm. Let

$$S_0 = n;$$
$$S_i = \left\lfloor S_{i-1} \frac{1 + U_i}{2} \right\rfloor \quad (i \geq 1),$$

where $U_1, U_2, \ldots$ are i.i.d. uniform [0, 1] random variables. The sequence $S_0, S_1, \ldots$ corresponds to the sizes of the subtrees rooted at the nodes processed on the first pass (see Fact 2).

It is clear that $L_n = k$ if and only if $S_{k-1} > 1$ and $S_k = 1$. The asymptotic behavior of L_n is covered by the following result.

FACT 3. *As* $n \to \infty$, *we have*

$$\frac{L_n}{\log n} \to \frac{1}{\log(e/2)} = 3.258891\ldots \quad a.s.$$

as $n \to \infty$. *In particular, if* $a < 1/\log(e/2)$, *then*

$$\mathbf{P}\{L_n < a \log n\} = O(n^{-\beta})$$

for some $\beta > 0$ *depending upon* a.

Proof. Observe that

$$n \prod_{i=1}^{k} \left(\frac{1+U_i}{2}\right) - k \le S_k \le n \prod_{i=1}^{k} \left(\frac{1+U_i}{2}\right).$$

Hence,

$$\begin{aligned}
\mathbf{P}\{L_n \le k\} &\le \mathbf{P}\{S_k \le 1\} \\
&\le \mathbf{P}\left\{n \prod_{i=1}^{k} \left(\frac{1+U_i}{2}\right) \le k+1\right\} \\
&\le \mathbf{P}\left\{\log n + \sum_{i=1}^{k} \log\left(\frac{1+U_i}{2}\right) \le \log(k+1)\right\} \\
&= \mathbf{P}\left\{\frac{1}{k}\sum_{i=1}^{k} \log\left(\frac{1+U_i}{2}\right) \le -\frac{\log n}{k} + \frac{\log(k+1)}{k}\right\}.
\end{aligned}$$

Now take $k = \lceil a \log n \rceil$ with $0 < a < 1/\log(e/2)$. We now apply a large deviation theorem (see, e.g., [22] or [12, §1.9]), which states that if $Y_1, Y_2, \ldots$ are i.i.d. random variables with mean μ, and if $\mathbf{E}e^{-tY_1} < \infty$ for some $t > 0$, then, for every $\epsilon > 0$, there exists a $\delta > 0$ such that

$$\mathbf{P}\left\{\frac{1}{k}\sum_{i=1}^{k} Y_i < \mu - \epsilon\right\} \le e^{-k\delta}.$$

Now, applied to our situation, noting that

$$\mathbf{E} \log\left(\frac{1+U_i}{2}\right) = \log(2/e)$$

and that

$$-\frac{\log n}{k} + \frac{\log(k+1)}{k} = -\frac{1}{a} + o(1),$$

we obtain

$$\mathbf{P}\left\{\frac{1}{k}\sum_{i=1}^{k} \log\left(\frac{1+U_i}{2}\right) \le -\frac{\log n}{k} + \frac{\log(k+1)}{k}\right\} = O(e^{-\delta \log n}) = O(n^{-\delta})$$

for some $\delta > 0$, provided that $-1/a < \log(2/e)$. This proves the second half of Fact 3. In the same manner, one can show that

$$\mathbf{P}\{L_n > a \log n\} = o(1)$$

for all $a > 1/\log(e/2)$. □

THEOREM. *The expected time taken by the algorithm given above is* $O(n/\log n)$.

Proof. The running time of the algorithm is appropriately measured by the number of nodes that are ever stacked. On the first pass just described, we push at most $2(L_n + 1)$ onto the stack. After this first pass, only nodes with the property that the (l, s) pair satisfies $l + s - 1 > L_n$ can ever be pushed onto the stack. The expected number of nodes pushed on the stack after the first pass is not more than

$$\begin{aligned}
&\mathbf{E}\left\{\min\left(n, \sum_{k=1}^{\infty} 2^k I_{[N_k \geq 1, k+N_k-1>L_n]}\right)\right\} \\
&\leq \sum_{k \geq a \log n} 2^k \mathbf{P}\{N_k \geq 1\} + \sum_{k < a \log n} 2^k \mathbf{P}\{a \log n + N_k - 1 > 3.2 \log n\} \\
&\quad + n\mathbf{P}\{L_n < 3.2 \log n\} \\
&\stackrel{\text{def}}{=} I + II + III ,
\end{aligned}$$

where N_k is the size of the subtree of a typical node at distance k from the root. Here we took into account that there are potentially 2^k nodes at distance k from the root. Observe next that

$$N_k \stackrel{\mathcal{L}}{=} \lfloor N_{k-1} U_k \rfloor , \quad N_0 = n,$$

where $U_1, U_2, \ldots$ are i.i.d. uniform [0, 1] random variables. Thus,

$$N_k \leq n \prod_{i=1}^{k} U_i \stackrel{\mathcal{L}}{=} e^{\log n - G_k},$$

where G_k is a gamma (k) random variable. We have

$$\mathbf{P}\{N_k \geq 1\} \leq \mathbf{P}\{\log n - G_k \geq 0\} \leq \frac{(\log n)^k e^{-\log n}}{k!(1 - \log n/(k+1))}$$

by an inequality for the left tail of the gamma distribution found, for example, in [9]. As a result of this estimate, we see that for $a > 1$,

$$\begin{aligned}
I &= \sum_{k \geq a \log n} 2^k \mathbf{P}\{N_k \geq 1\} \\
&\leq \sum_{k \geq a \log n} \frac{(2 \log n)^k e^{-\log n}}{k!(1 - 1/a)} \\
&= \frac{a}{a-1} e^{\log n} \sum_{k \geq a \log n} \frac{(2 \log n)^k e^{-2 \log n}}{k!} \\
&= \frac{an}{a-1} \mathbf{P}\{Z \geq a \log n\} ,
\end{aligned}$$

where Z is a Poisson $(2 \log n)$ random variable. We will see that we need to take $a \in (2, 3.2)$. By Chebyshev's inequality, since $a > 2$, we see that

$$\mathbf{P}\{Z \geq a \log n\} \leq \frac{\text{Var}\{Z\}}{(a-2)^2 \log^2 n} = \frac{2}{(a-2)^2 \log n} .$$

We conclude that for $a > 2$,

$$I \leq \frac{2an}{(a-1)(a-2)^2 \log n}.$$

Next,

$$III = n\mathbf{P}\{L_n < 3.2 \log n\} \leq n^{1-\delta}$$

for some $\delta > 0$ by Fact 3. Finally, if $y = \log n - \log\log n - \log(3.2 - a)$,

$$\begin{aligned}
II &= \sum_{k<a\log n} 2^k \mathbf{P}\{a\log n + N_k - 1 > 3.2\log n\} \\
&\leq \sum_{k<1.1\log n} 2^{1+1.1\log n} + \sum_{1.1\log n \leq k < a\log n} 2^k \mathbf{P}\{N_k - 1 > (3.2-a)\log n\} \\
&\leq 2n^{0.77} + \sum_{1.1\log n \leq k < a\log n} 2^k \mathbf{P}\{\log n - G_k > \log(3.2-a) + \log\log n\} \\
&\leq 2n^{0.77} + \sum_{1.1\log n \leq k < a\log n} 2^k \frac{y^k e^{-y}}{k!(1 - y/(k+1))} \\
&\leq 2n^{0.77} + 11\, e^y \sum_{1.1\log n \leq k < a\log n} \frac{(2y)^k e^{-2y}}{k!} \\
&\leq 2n^{0.77} + 11\, e^y \\
&= 2n^{0.77} + \left(\frac{11}{3.2-a}\right)\frac{n}{\log n}.
\end{aligned}$$

Thus, $I + II + III = O(n/\log n)$. □

4. A simple linear time algorithm. In [7, p.650], a linear time method is given for generating a random binary search tree. The basic algorithm is shown below.

```
m ← 0 (m is the number of levels)
n_0 ← 1
for N := 1 to n do (N is the number of external nodes)
  generate L randomly in {0, ..., m}
    according to the frequencies n_0, ..., n_m
  n_L ← n_L − 1
  if L = m then m ← m + 1
  n_{L+1} ← n_{L+1} + 2
return (n_0, n_1, ..., n_m)
(the height of the tree is m − 1)
```

In this algorithm, we keep the number of external nodes at each level in an array $(n_0, n_1, \ldots, n_m)$. The expected storage needed for this is $O(\log n)$ since the expected height is $O(\log n)$. The algorithm is based upon the fact that when adding a new node, each of the external nodes is equally likely to receive the node. To generate the random integer L according to the vector of frequencies $(n_0, \ldots, n_m)$, one can trivially proceed in time $O(1 + m)$, but

this would result in overall expected time $O(n \log n)$. To generate L in $O(1)$ expected time, one has to either use more space or more programming resources. For example, keeping an array with $n_0 + n_1 + \cdots + n_m$ entries, of which n_i entries have label i, would enable us to generate L in $O(1)$ time. The overall time and space both are $O(n)$. By a dynamic version of the guide table method (see [3] or [13], for the raw guide table method), $O(\log n)$ expected space is achievable provided that we can update the guide table in $O(1)$ amortized time. This is easy to achieve if we take care to rebuild the table every $\log n$th (or mth) entry. Between rebuilding, the new entries in the table are collected in a simple overflow list; this does not affect the overall linear expected time.

5. Discrete jumps in the simulation: An $O(\sqrt{n} \log n)$ method.

5.1. Description. Consider a vector $(n_0, n_1, \ldots, n_m)$ representing the number of external nodes at levels $0, 1, \ldots, m$, respectively, and assume that $n_0 + \cdots + n_m = n$ for now. The previous linear time algorithm is based upon the selection of a uniform random external node, say one at level $k \in \{0, \ldots, m\}$, and the updating of the vector by the rule

$$\begin{cases} n_k \leftarrow n_k - 1, \\ n_{k+1} \leftarrow n_{k+1} + 2. \end{cases}$$

Imagine that the n original external nodes are white balls in an urn, and that the label of each ball is its level number. A randomly selected (white) ball is removed. If its label is k, two black balls with label $k + 1$ are added. This process can be repeated until we pick a black ball for the first time. The number of draws required until this happens is a random variable $T_n \in \{2, \ldots, n + 1\}$. We say that T_n has the *waiting-time distribution* with parameter n. We can let the tree-growing process jump ahead by T steps at once if we are given T. Indeed, given T, it suffices to draw $T - 1$ white balls uniformly and without replacement from the urn. The vector $(D_0, D_1, \ldots, D_m)$ represents the number of balls drawn with labels $0, 1, \ldots, m$, respectively. The distribution of this vector is multivariate hypergeometric; the details on how to generate the vector on a computer will be given later; it suffices for now to say that this vector can be generated in $O(m)$ expected time uniformly over all n. Now, the vector of external nodes is updated by the rule

$$\begin{aligned} (n_0, \ldots, n_m, n_{m+1}) \leftarrow\ & (n_0, \ldots, n_m, 0) \\ & - (D_0, D_1, \ldots, D_m, 0) \\ & + 2(0, D_0, \ldots, D_m)\ . \end{aligned}$$

The single black ball is taken care of by selecting a label L at random from the $T - 1$ white ball labels just selected, the kth label being picked with probability proportional to D_k. This label can be chosen in time $O(m)$ by the trivial algorithm

```
generate X uniformly on {1,...,T − 1}.
S ← D_0, L ← 0.
while X > S do
  L ← L + 1
  S ← S + D_L
return L
```

A further update is required:

$$(n_{L+1}, n_{L+2}) \leftarrow (n_{L+1} - 1, n_{L+2} + 2)\ ,$$

where, if $L = m + 1$, we define $n_{m+2} = 0$ before the update and $n_{m+2} = 2$ after the change. The number of external nodes now is $n + T$ instead of n. Iterate this process until we obtain more external nodes than needed. It is a simple matter to get the exact size one wants by simply truncating T in the last iteration. Let us first provide the algorithm in its entirety:

```
N ← 1 (N is the number of external nodes)
m ← 0 (m is the number of levels)
n_0 ← 1
repeat
  generate T ∈ {2, ..., N + 1} with the waiting time distribution
    with parameter N
  if T + N > n + 1
    then T ← n + 2 − N, S ← −1
    else generate S uniformly in {0, ..., T − 1}
  generate a multivariate hypergeometric (D_0, ..., D_m)
    with parameters T − 1 and (n_0, ..., n_m)
if D_m > 0
  then m ← m + 1
       (n_0, ..., n_m) ← (n_0, ..., n_{m−1}, 0) − (D_0, ..., D_{m−1}, 0) + 2(0, D_0, ..., D_{m−1})
  else (n_0, ..., n_m) ← (n_0, ..., n_m) − (D_0, ..., D_m) + 2(0, D_0, ..., D_{m−1})
if S ≠ −1
  then generate a random integer L ∈ {0, ..., m}
       from S by inversion according to the frequencies D_0, ..., D_m
       n_{L+1} ← n_{L+1} − 1
       if L + 2 ≤ m then n_{L+2} ← n_{L+2} + 2
                    else n_{L+2} ← 2, m ← m + 1
until T + N ≥ n + 1
return (n_0, n_1, ..., n_m)
```

The algorithm given above returns a vector with m components n_i, where n_i is the number of external nodes on level i. Clearly, for a random binary search tree with n nodes, we have $n + 1$ external nodes, and thus $\sum_i n_i = n + 1$. Thus, besides the height of the binary search tree $(m - 1)$, we also have information about the distribution of the nodes over the various levels.

A few details have to be ironed out:

- Determine the waiting time distribution for T and show how a random variate T can be generated in constant expected time, bounded uniformly over n.
- Show how one can generate a multivariate hypergeometric random variate.
- Show that the algorithm takes $O(\sqrt{n} \log n)$ expected time units.

5.2. The distribution of *T*. By using the urn with white balls and black balls, we see that $\mathbf{P}\{T > k\}$ is equal to the probability that in the first k draws only white balls are chosen. Clearly then, if the urn has n white balls to begin with,

$$\mathbf{P}\{T > k\} = \prod_{i=1}^{k-1} \frac{n-i}{n+i}, \quad 2 \le k \le n+1$$

so that

$$\mathbf{P}\{T = k+1\} = \frac{2k}{n} \frac{\binom{2n}{n+k}}{\binom{2n}{n}}, \quad 1 \le k \le n.$$

Random variate generation can be dealt with by von Neumann's rejection method, which requires a summable function of k that dominates the probability vector given above. In von Neumann's method, worked out below for our case, it suffices to generate random variates from a distribution with probabilities proportional to the bounding vector and stop when for the first time an acceptance condition is satisfied. We first derive an upper bound from the following inequality:

$$\mathbf{P}\{T = k+1\} \le \frac{2k}{n} e^{-2k^2/3n} , \quad 1 \le k \le n .$$

This can be shown as follows. Using $\log(1+u) \ge 2u/(2+u)$ for $u > 0$, we have

$$\begin{aligned}
\mathbf{P}\{T = k+1\} &\le \frac{2kn^{k-1}}{(n+k)^k} \le \frac{2k}{n} e^{-2k^2/(2n+k)} \\
&\le \frac{2k}{n} e^{-2k^2/3n} \\
&\le \frac{2(x+1)}{n} e^{-2x^2/3n} , \quad k-1 < x \le k .
\end{aligned}$$

Observe that

$$\int_0^\infty \frac{2x}{n} e^{-2x^2/3n} = \frac{3}{2} ;$$

$$\int_0^\infty \frac{2}{n} e^{-2x^2/3n} = \sqrt{\frac{3\pi}{2n}} .$$

Thus, the density $\frac{2(x+1)}{n} e^{-2x^2/3n}$ on the positive halfline is the mixture of the Maxwell density and the normal density. The rejection method for T can be summarized as follows:

```
generator for T with parameter n

repeat
  generate U, V uniform [0, 1]
  if U < 3/2 / (3/2 + sqrt(3π/2n))
    then generate an exponential random variate E; set Y ← sqrt(3nE/2)
    else generate N standard normal; set Y ← sqrt(3nN²/4)
  X ← ⌈Y⌉
until Y ≤ n and V (2X+2)/n exp(−2X²/3n) < (2Y/n) ( C(2n, n+Y) / C(2n, n) )
return T ← Y + 1
```

The expected number of iterations in this algorithm is $3/2 + \sqrt{3\pi/4n}$. The expected time is uniformly bounded over n if we can evaluate factorials in $O(1)$ time or if we can verify the acceptance condition in $O(1)$ time. If we accept a model in which simple basic operations take constant time, regardless of the size of the operands. The factorial, evaluated naively, would thus take time proportional to n. One can consult Chapter X of [7] on this issue; depending upon the situation, one can use a combination of either Stirling's approximation or Binet's approximation with the alternating series acceptance/rejection method. Another property we will require is that $\mathbf{P}\{T > \sqrt{n/4}\} \ge 1/2 > 0$. To see this, observe that with $s = \lceil \sqrt{n/4} \rceil \le n$, for $s - 1 \ge 1$,

$$\begin{aligned}\mathbf{P}\{T > s\} &\geq \left(\frac{n-s+1}{n+s-1}\right)^{s-1} \\ &= \left(1 - \frac{2s-2}{n+s-1}\right)^{s-1} \\ &\geq \left(1 - \frac{2(s-1)^2}{n+s-1}\right) \\ &\geq \frac{1}{2}\end{aligned}$$

since $s \leq 1 + \sqrt{n/4}$. When $s = 1$ (thus, $n \leq 4$), the inequality we are trying to establish is obviously satisfied, as $T \geq 2$ with probability one.

5.3. Generation of random vectors with a multivariate hypergeometric distribution. The random vector $(D_0, D_1, \ldots, D_m)$ obtained by drawing without replacement k balls from an urn having n_i balls with label i, $0 \leq i \leq m$, is called a multivariate hypergeometric random vector with parameters k and $(n_0, n_1, \ldots, n_m)$. In a simple hypergeometric situation, we have $m = 1$. For $m = 1$, various generators have uniformly fast expected time per random variate; see for example the generators of Kachitvichyanukul [15], Kachitvichyanukul and Schmeiser [16], [17], Stadlober [29]-[31], or Devroye [7]. Using these algorithms, we can generate D_0 by drawing from an urn with n_0 balls labeled 0 and $n_1 + \cdots + n_m$ balls labeled "> 0." By repeating this step, we can generate the multivariate hypereometric random variate in expected time $O(m)$ uniformly in all the other parameters.

5.4. Probabilistic analysis. We will show the following fact.

FACT 4. *The algorithm given above takes expected time* $O(\sqrt{n}\log n)$.

Proof. The following notation will be used: the number of external nodes at the beginning of the ith iteration is N_i: thus,

$$1 = N_0 < N_1 < N_2 < \cdots .$$

We consider the algorithm without truncation, iterated ad infinitum. Let T_i be the value of the waiting time random variable for the ith iteration; its parameter is N_{i-1}, and we have

$$N_i = N_{i-1} + T_i \ (i \geq 1) .$$

The algorithm halts after iteration k when for the first time $N_k \geq n + 1$. The number of iterations is denoted by J_n. Clearly, $J_n > k$ if and only if $N_k < n + 1$. Also, by construction, $N_i \geq 2i$ for all i, so that $J_n \leq n$. This implies that

$$\mathbf{E}J_n^2 \leq \sum_{1 \leq i \leq n} \mathbf{P}\{J_n^2 \geq i\} \leq \inf_k \left(n^2 \mathbf{P}\{N_k < n+1\} + k^2\right).$$

We will see that $\mathbf{E}J_n^2 = O(n)$. Then we continue as follows: if the vector of external nodes at the outset of an iteration is $(n_0, \ldots, n_m)$, then that iteration takes expected time bounded by a universal constant times $m + 1$. Clearly, $m \leq H_n$, the height of the random tree generated. Hence, the overall expected time is bounded by

$$c\mathbf{E}\{J_n + J_n H_n\},$$

where c is a universal constant. By the Cauchy-Schwarz inequality, we have

$$\mathbf{E}\{J_n H_n\} \leq \sqrt{\mathbf{E}J_n^2 \mathbf{E}H_n^2} .$$

From [6], we recall that $\mathbf{E}H_n^2 = O(\log^2 n)$. Since $\mathbf{E}J_n^2 = O(n)$, Fact 4 is proved.

To show $\mathbf{E}J_n^2 = O(n)$, it clearly suffices to choose $k = \Theta(\sqrt{n})$ and to show that

$$\mathbf{P}\{N_k < n+1\} = O(1/n) .$$

Let I_i be the indicator function of the event $T_i > \sqrt{N_{i-1}/4}$. Thus,

$$N_i \geq N_{i-1} + \max\left\{2, \sqrt{N_{i-1}/4}\, I_i\right\}, \quad i \geq 1 .$$

Consider a deterministic sequence d_i determined by $d_0 = 1$,

$$d_i = d_{i-1} + \max\left\{2, \sqrt{d_{i-1}/4}\right\} \quad (i \geq 1) .$$

By induction, it is easy to see that $N_i \geq d_{B_i}$, where

$$B_i = \sum_{j=1}^{i} I_j .$$

Again by induction, we have $d_i \geq (i/8)^2$ for all $i \geq 1$. Thus, $N_i \geq (B_i/8)^2$. The I_j's are dependent. Nevertheless, we have shown that $\mathbf{E}\{I_j | I_1, \ldots, I_{j-1}\} \geq 1/2$, so that by a simple coupling argument, B_i is stochastically dominated by a binomial $(i, 1/2)$ random variable B_i'. Therefore, by Hoeffding's inequality [14],

$$\begin{aligned}
\mathbf{P}\{N_k < n+1\} &\leq \mathbf{P}\{B_k < 8\sqrt{n+1}\} \\
&\leq \mathbf{P}\left\{B_k - \mathbf{E}B_k < 8\sqrt{n+1} - k/2\right\} \\
&\leq \mathbf{P}\{B_k - \mathbf{E}B_k < -k/4\} \\
&\qquad (\text{if } k \geq 32\sqrt{n+1}) \\
&\leq e^{-2(k/4)^2/k} \\
&= e^{-k/8} .
\end{aligned}$$

Take $k = \lceil 32\sqrt{n+1} \rceil$ and conclude that

$$\mathbf{P}\{N_k < n+1\} = O(1/n),$$

as required. In fact, we have shown that

$$\mathbf{E}J_n^2 \leq 16n + 17$$

for all n large enough. □

6. A birth-and-death process method.

6.1. Derivation. In Robson [26], [27], simulations were reported that were based upon an ultra-fast algorithm that produces random binary search trees of random size. This method has never been published nor analyzed. Also, the modifications required to produce a random tree of the correct size are discussed in this paper.

Once again, consider a vector $(n_0, n_1, \ldots, n_m)$ representing the number of external nodes at levels $0, 1, \ldots, m$, respectively, and assume that $n_0 + \cdots + n_m = n$ for now. Every external node should be considered as a living element in a birth-and-death process with unit reproduction rate for each element. When an external node gives birth, it produces two new elements (which live at one level below their parent), and it immediately dies, for a net gain of one element. This is nothing but the Yule process (a special case of a pure birth process; see [32, p. 215]). The n nodes at time t will thus spawn families of offspring at time $t + \Delta$ of i.i.d. sizes $S_1, \ldots, S_n$. The common distribution of the S_i's is that of S, where

$$\mathbf{P}\{S = k\} = \left(1 - e^{-\Delta}\right)^{k-1} e^{-\Delta} \ (k \geq 1) \ .$$

If the state at time t is described by $(n_0, n_1, \ldots, n_m)$, then our purpose is to efficiently generate an updated state at time $t + \Delta$, where Δ is a constant to be selected. The first step is to generate the sizes of the subtrees of external nodes (at time $t + \Delta$) with roots at the elements alive at time t. This leads to the generation of the triangular array of random integers $N(i, j)$, each representing the number of size j subtrees with original root at level i. Thus,

$$\sum_{j=1}^{\infty} N(i, j) = n_i \ , \ 0 \leq i \leq m \ .$$

In fact, $(N(i, 1), N(i, 2), N(i, 3), \ldots)$ is multinomial with parameter n_i and probabilities given by $p_1(\Delta), p_2(\Delta), \ldots$. By repeatedly appealing to a uniformly fast binomial generator, we can generate this vector in expected time bounded by a constant times the expected value of the maximal size subtree $M(n_i)$ for any of the n_i roots. Now, the maximum of n_i i.i.d. geometric random variables described by the probabilities $p_i(\Delta), i \geq 1$, has an expected value not exceeding

$$1 + \frac{1 + \log n_i}{\log(1/(1 - e^{-\Delta}))} \ .$$

Since each n_i does not exceed n, we see that, given m, all $N(i, j)$ can be generated in expected time $O(m \log n)$.

The next step in the algorithm consists of generating the correct numbers of external nodes at all levels. This can be done in one sweep from 0 to m. Assume that we are given $N(i, j)$, $j \geq 1$, for fixed i. This leads to $N(i, 1)$ external nodes at level i. For fixed $j \geq 2$, we obtain no external nodes at level i. Rather, it is possible to determine how many subtrees rooted at nodes of level $i + 1$ this leads to. Indeed, a node at level i with a subtree having j external nodes yields a left child at level $i + 1$ which itself has a subtree of (random) size $S \geq 2$, where S is equally likely to take any value between 1 and $j - 1$. The size of the subtree rooted at the right child is $j - S$. Of course, we won't have to do this for each node separately. Rather, it is easy to see that we need only generate a multinomial random vector $w_1, w_2, \ldots, w_{j-1}$ with $\sum w_l = N(i, j)$, and equal probabilities. Here w_l represents the number of left child nodes at level $i + 1$ having l external nodes in their subtrees. Adding in the sizes of the right subtrees, we see that at level $i + 1$, the $N(i, j)$ level i nodes spawn $2N(i, j)$ nodes with $w_l + w_{j-l}$ nodes of size l. A uniformly fast binomial generator can "split" all $N(i, j)$ at level i in this manner in expected time $O(\mathbf{E}M(n_i)^2)$. For fixed Δ, this is $O(\log^2 n)$.

The sizes of the nodes at level $i+1$ can now be updated, and they in turn are split. An iteration thus takes expected time not exceeding $\mathbf{E}H_N$ times $O(\log^2 n)$ where N is the number of external nodes at the end of the iteration.

```
a generator for a random binary search tree
  with ≥ n + 1 external nodes

N ← 1, m ← 0, n_0 ← 1
while N < n + 1 do
  r ← 1, t_1 ← 0
  processed ← 0, j ← 0
  while processed < N do
    generate a multinomial random vector (u_1, u_2, ..., u_R) with
      parameter n_j and probabilities p_k = e^{-Δ}(1 - e^{-Δ})^{k-1}, k ≥ 1
  N ← N + u_1 + 2u_2 + ··· + Ru_R - n_j
  if R < r then (u_{R+1}, ..., u_r) ← 0, R ← r
  (u_1, u_2, ..., u_r) ← (u_1, u_2, ..., u_r) + (t_1, t_2, ..., t_r)
  n_j ← u_1, processed ← processed + u_1
  j ← j + 1
  m ← 2, (t_1, ..., t_R) ← 0
  while m ≤ R do
    generate a multinomial random vector (w_1, ..., w_{m-1})
      with parameter u_m and equal probabilities 1/(m - 1)
    (t_1, ..., t_{m-1}) ← (t_1, ..., t_{m-1}) + (w_1, ..., w_{m-1})
    (t_1, ..., t_{m-1}) ← (t_1, ..., t_{m-1}) + (w_{m-1}, ..., w_1)
    m ← m + 1
  if Σ_2^R u_m = 0 then r ← 1 else r ← max{i : t_i > 0}
return (n_0, n_1, ..., n_m)
```

6.2. Probabilistic analysis.

FACT 5. *The algorithm shown above halts in expected time* $O(\log^4 n)$.

Proof. Consider the overall number of external nodes T_i after i iterations, starting with $T_0 = 1$. T_i is the sum of T_{i-1} i.i.d. geometric random variables with probabilities given by $p_1(\Delta), p_2(\Delta), \ldots$. The expected value of one geometric random variable is e^{Δ}. Thus,

$$\mathbf{E}T_i = e^{\Delta}\mathbf{E}T_{i-1} = e^{i\Delta} \ (i \geq 0) \ .$$

The tree thus grows exponentially quickly. In fact, we know much more. By the derivation given above, the distribution of T_i is geometric itself with parameter $e^{-i\Delta}$. It is perhaps a bit counterintuitive that T_i has a monotonically decreasing discrete density, with the value 1 being most likely. Clearly, if we are aiming for a tree of approximate size n, then we can take $\Delta = \log 2$, and perform $\log_2 n$ iterations. Or we can iterate until for the first time $T_i > n+1$. In the latter case, if the number of iterations is J_n and the height after stopping is H_N, the overall expected time is bounded by $O(\log^2 n)$ times

$$\mathbf{E}\{J_n H_N\} \leq \sqrt{\mathbf{E}\left\{J_n^2\right\}\mathbf{E}\left\{H_N^2\right\}} = O(\log^2 n),$$

provided that $\mathbf{E}H_N = O(\log n)$ and that $\mathbf{E}\left\{J_n^2\right\} = O(\log^2 n)$. We conclude that the expected complexity is $O(\log^4 n)$. But

$$\mathbf{P}\{J_n > k\} \leq \mathbf{P}\{T_k \leq n+1\} = 1 - \left(1 - e^{-k\Delta}\right)^{n+1}$$

so that

$$\begin{aligned}
\mathbf{E}\left\{J_n^2\right\} &= \sum_k 2k\mathbf{P}\{J_n > k\} \\
&\le k^*(k^*+1) + (n+1)\sum_{k>k^*} 2ke^{-k\Delta} \\
&\le k^*(k^*+1) + (n+1)\int_{k^*}^{\infty} 2xe^{-x\Delta}\,dx \\
&= k^*(k^*+1) + (n+1)2\Delta^{-2}(1+\Delta k^*)e^{-k^*\Delta} \\
&\le \left(2+\frac{\log n}{\Delta}\right)^2 + 2\Delta^{-2}(1+\log(n+1)) \\
&\qquad \text{(take } k^* = \lceil 2\log(n+1)/\Delta\rceil) \\
&= O(\log^2 n)\,.
\end{aligned}$$

Finally, we show that $\mathbf{E}H_N = O(\log n)$. Clearly, for $k > 0$,

$$\begin{aligned}
&\mathbf{P}\{H_N > 4e\log(n+1) + 4ek + 1\} \\
&\quad \le \mathbf{P}\{N > k^2(n+1)^2\} + \mathbf{P}\{H_{k^2(n+1)^2} > 4e\log(n+1) + 4ek + 1\},
\end{aligned}$$

where we used the monotonicity of the tree-building process; here H_j denotes the height of the tree in the birth-and-death process at the moment that it has precisely $j+1$ external nodes. The last probability decreases exponentially quickly with n and k: for $k \ge 2\log n$,

$$\mathbf{P}\{H_n \ge k\} \le \frac{2(2\log n)^k}{n\,k!} \le \frac{2(2e\log n/k)^k}{n}$$

[9, Thm 5]. Therefore,

$$\begin{aligned}
&\mathbf{P}\{H_{k^2(n+1)^2} > 4e\log(n+1) + 4ek + 1\} \\
&\quad \le 2\left(\frac{4e(\log(n+1)+\log k)}{4e\log(n+1)+4ek}\right)^{(4e\log(n+1)+4ek+2}} \times \frac{1}{k^2(n+1)^2} \\
&\quad \le 2k^{-2}(n+1)^{-2}\,.
\end{aligned}$$

Also,

$$\begin{aligned}
\mathbf{P}\{N > k^2(n+1)^2\} &\le \mathbf{P}\{\cup_{k=0}^n [T_k < n+1,\, T_{k+1} \ge k^2(n+1)^2]\} \\
&\le (n+1)\sup_{1\le k\le n} \mathbf{P}\{T_k < n+1,\, T_{k+1} \ge k^2(n+1)^2\} \\
&= (n+1)\sup_{1\le k\le n} \sum_{j=1}^{n} \mathbf{P}\{T_k = j\}\mathbf{P}\{T_{k+1} \ge k^2(n+1)^2 | T_k = j\} \\
&\le (n+1)\sup_{1\le k\le n} \mathbf{P}\{T_{k+1} \ge k^2(n+1)^2 | T_k = n\} \\
&= (n+1)\mathbf{P}\{T_1 \ge k^2(n+1)^2 | T_0 = n\} \\
&\le k^{-4}(n+1)^{-3}\mathbf{E}\{T_1^2 | T_0 = n\} \\
&= k^{-4}(n+1)^{-3}\left((ne^{\Delta})^2 + n(e^{\Delta}-1)\right) \\
&\le \frac{2e^{2\Delta}}{k^4(n+1)}\,.
\end{aligned}$$

Sum the upper bounds over $k > 0$, and conclude that $\mathbf{E}H_N \le 4e\log(n+1) + O(1)$. □

6.3. Stopping at exactly *n* nodes. To obtain a tree of exactly n nodes from the algorithm above, we need to modify it so that before each iteration copies of N, m and $(n_0, n_1, \ldots, n_m)$ are made and, if the iteration would increase the number of nodes beyond n, the iteration is aborted by returning to the copied values. To preserve the expected time $O(\log^4 n)$, it is also necessary to reduce the value of Δ when it is likely that taking a time step of Δ will give an aborted iteration. A simple approach is to take Δ as $\log((n+N)/2N)$ whenever this is less than $\log 2$ (that is when $N > n/3$), so that the expected number of new nodes created is half of the number required to reach n.

FACT 6. *The algorithm modified in this way halts in expected time* $O(\log^4 n)$.

Proof. To prove the bound $O(\log^4 n)$ on the expected time, it is only necessary to show that the bound $\mathbf{E}\{J_n^2\} = O(\log^2 n)$ still holds. First we count only nonaborted iterations. By the reasoning above, the bound holds for the number of iterations to reach $n/3$ nodes. After $n/3$ nodes, each iteration has value of Δ such that the expected number of new nodes is half that required to reach n and it is easy to see that there is a constant $c > 0$ such that each iteration has probability at least c of adding at least half of that expected number (since the rate of creation of new nodes is at least that of a Poisson process creating N nodes per unit time). But the maximum number of iterations after $n/3$ nodes which add at least a quarter of the $n-N$ nodes still required is $O(\log n)$ so that the expectation $\mathbf{E}\{J_n^2\} = O(\log^2 n/c^2) = O(\log^2 n)$ as required.

To handle aborted iterations, we use a similar argument: each iteration has a probability at least $1/2$ of not being aborted; hence including aborted iterations can increase $\mathbf{E}\{J_n^2\}$ by at most a factor of 4. □

In practice it would be sensible to halt this process when the number of nodes required to be added was small (say $< \sqrt{n}\log^2 n$) and finish the simulation with the method of §5.

Acknowledgments. We thank the referees.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.

[2] J. H. AHRENS AND U. DIETER, *Sampling from binomial and Poisson distributions: A method with bounded computation times*, Computing, 25 (1980), pp. 193-208.

[3] H. C. CHEN AND Y. ASAU, *On generating random variates from an empirical distribution*, AIIE Transactions, 6 (1974), pp. 163-166.

[4] T. H. CORMEN, C. E. LEISERSON, AND R. L. RIVEST, *Introduction to Algorithms*, MIT Press, Boston, 1990.

[5] L. DEVROYE, *Generating the maximum of independent identically distributed random variables*, Comput. Math. Appl., 6 (1980) pp. 305-315.

[6] ———, *A note on the height of binary search trees*, J. Assoc. Comput. Mach., 33 (1986), pp. 489-498.

[7] ———, *Non-Uniform Random Variate Generation*, Springer-Verlag, New York, 1986.

[8] ———, *A simple generator for discrete log-concave distributions*, Computing, 39 (1987), pp. 87-91.

[9] ———, *Branching processes in the analysis of the heights of trees*, Acta Inform., 24 (1987), pp. 277-298.

[10] ———, *A simple generator for discrete log-concave distributions*, Computing, 39 (1987), pp. 87-91.

[11] ———, *Generating sums in constant average time*, in Proceedings of the 1988 Winter Simulation Conference, M. A. Abrams, P. L. Haigh, and J. C. Comfort, eds., IEEE, San Diego, CA., 1988, pp. 425-431.

[12] R. DURRETT, *Probability: Theory and Examples*, Wadsworth and Brooks, Pacific Grove, CA, 1991.

[13] G. S. FISHMAN AND L. R. MOORE, *Sampling from a discrete distribution while preserving monotonicity*, Amer. Statist., 38 (1984), pp. 219-223.

[14] W. HOEFFDING, *Probability inequalities for sums of bounded random variables*, J. Amer. Statist. Assoc., 58 (1963), pp. 13-30.

[15] V. KACHITVICHYANUKUL, Computer Generation of Poisson, Binomial, and Hypergeometric Random Variates, Ph.D. Dissertation, School of Industrial Engineering, Purdue University, West Lafayette, IN, 1982.

[16] V. KACHITVICHYANUKUL AND B. W. SCHMEISER, *Computer generation of hypergeometric random variates*, J. Statist. Comput. Simulation, 22 (1985), pp. 127-145.

[17] V. KACHITVICHYANUKUL AND B. W. SCHMEISER, *Binomial random variate generation,* Comm. ACM, 31 (1988), pp. 216-222.

[18] ———, *Algorithm* 668 *H2PEC: Sampling from the hypergeometric distribution,* ACM Trans. Math. Software, 14 (1988), pp. 397-398.

[19] D. E. KNUTH, *The Art of Computer Programming, Vol.* 3*: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[20] H. M. MAHMOUD, *Evolution of Random Search Trees*, John Wiley, New York, 1992.

[21] H. MAHMOUD AND B. PITTEL, *On the most probable shape of a search tree grown from a random permutation,* SIAM J. Alg. Discrete Methods, 5 (1984), pp. 69-81.

[22] V. V. PETROV, *Sums of Independent Random Variables*, Springer-Verlag, Berlin, 1975.

[23] B. PITTEL, *On growing random binary trees,* J. Math. Anal. App. 103 (1984), pp. 461-480.

[24] B. B. POKHODZEI, *Beta- and gamma-methods of modelling binomial and Poisson distributions,* USSR Computational Mathematics and Mathematical Physics, 24 (1984), pp. 114-118.

[25] ———, *A note on beta-methods for simulating binomial distributions,* USSR Computational Mathematics and Mathematical Physics, 28 (1988), pp. 207-208.

[26] J. M. ROBSON, *The height of binary search trees,* The Australian Computer Journal, 11 (1979), pp. 151-153.

[27] ———, *The asymptotic behaviour of the height of binary search trees,* Austral. Comput. Sci. Comm., 1982, p. 88.

[28] B. W. SCHMEISER, *Generation of the maximum (minimum) value in digital computer simulation,* J. Statist. Comput. Simulation, 8 (1978), pp. 103-115.

[29] E. STADLOBER, *Sampling from Poisson, binomial and hypergeometric distributions: Ratio of uniforms as a simple fast alternative*, Habilitationsschrift, Institute of Statistics, Technical University of Graz, Austria, 1988.

[30] ———, *Binomial random variate generation: a method based on ratio of uniforms,* Amer. J. Math. Management Sci., 9 (1989), pp. 1-20.

[31] ———, *Ratio of uniforms as a convenient method for sampling from classical discrete distributions,"* in Proceedings of the 1989 Winter Simulation Conference, E. A. MacNair, K. J. Musselman, and P. Heidelberger, eds., IEEE, Washington, DC, 1989, pp. 484-489.

[32] H. M. TAYLOR AND S. KARLIN, *An Introduction to Stochastic Modeling*, Academic Press, Orlando, FL, 1984.

SIAM J. COMPUT.
Vol. 24, No. 6, pp. 1157-1162, December 1995

ON THE VARIANCE OF THE HEIGHT OF RANDOM BINARY SEARCH TREES*

LUC DEVROYE† AND BRUCE REED†

Abstract. Let H_n be the height of a random binary search tree on n nodes. We show that there exists a constant $\alpha = 4.31107\ldots$ such that $\mathbf{P}\{|H_n - \alpha \log n| > \beta \log\log n\} \to 0$, where $\beta > 15\alpha/\ln 2 = 93.2933\ldots$. The proof uses the second moment method and does not rely on properties of branching processes. We also show that $\mathrm{Var}\{H_n\} = O((\log\log n)^2)$.

Key words. binary search tree, probabilistic analysis, random tree, asymptotics, height, second moment method

AMS subject classifications. 68Q25, 60C05

1. Introduction.

The height H_n of a random binary search tree on n nodes, constructed in the usual manner, starting from a random equiprobable permutation of $1, \ldots, n$, is known to be close to $\alpha \log n$, where $\alpha = 4.31107\ldots$ is the unique solution on $[2, \infty)$ of the equation $\alpha \log((2e)/\alpha) = 1$. First, Pittel [12] showed that $H_n/\log n \to \gamma$ almost surely as $n \to \infty$ for some positive constant γ. This constant was known not to exceed α (Robson [15]), and it was shown in Devroye [4] that $\gamma = \alpha$ as a consequence of the fact that $\mathbf{E}H_n \sim \alpha \log n$. Robson [16] has found that H_n does not vary much from experiment to experiment and seems to have a fixed range of width not depending on n. Devroye [5] proved that $H_n - \alpha \log n = O(\sqrt{\log n \log\log n})$ in probability, but this does not quite confirm Robson's findings. It is the purpose of this paper to prove the following theorem.

THEOREM.

$$\mathbf{E}H_n = \alpha \log n + O(\log\log n)$$

and

$$\mathrm{Var}\{H_n\} = O((\log\log n)^2) .$$

While this is a major step forward, we still do not know whether $\mathrm{Var}\{H_n\} = O(1)$. For more information on random binary search trees, one may consult Knuth [7], [8], Aho, Hopcroft, and Ullman [1], [2], Mahmoud and Pittel [10], Devroye [6], Mahmoud [9], and Pittel [13].

Finally, we note that this paper contains the first proof of the asymptotic properties of H_n that is not based upon the theory of branching processes or branching random walks. We merely employ a well-known representation of random binary search trees from Devroye [4], and combine it with the second moment method, which has found so many other applications in the theory of random graphs (see, e.g., Palmer [11]).

2. Notation and definitions.

Let T_∞ be the complete infinite binary tree. Each node x has a right son $r(x)$ and a left son $l(x)$. We consider a random labelled tree R_∞ obtained from T_∞ by choosing a uniform [0, 1] random variable $U(x)$ for each node x of T_∞ and labelling the edge $(x, r(x))$ by $U(x)$ and the edge $(x, l(x))$ by $1 - U(x)$. The label of edge a is denoted $L(a)$. We let R_k be the random tree consisting of the first k edge levels of R_∞.

For each node y of R_∞, we let $f(y)$ be the product of the labels of the edges on the unique path from the root to y. We remark that for each $x \in R_\infty$, $-\log U(x)$ is an exponential

*Received by the editors September 24, 1992; accepted for publication (in revised form) April 21, 1994. This research was supported by Natural Sciences and Engineering Research Council of Canada grant A3456.

†School of Computer Science, McGill University, Montreal, Quebec H3A 2K6, Canada (luc@crodo.cs.mcgill.ca).

random variable with mean 1. If the labels on the path from the root to a node y of R_∞ are $U_1, \ldots, U_i$, then we define

$$h_n(y) = \lfloor \ldots \lfloor \lfloor nU_1 \rfloor U_2 \rfloor \ldots U_i \rfloor .$$

Also, $-\log f(y)$ is distributed as the sum of i independently and identically distributed (i.i.d.) exponential random variables with mean 1, i.e., it is gamma distributed with parameter i.

Fact 1. It is well known that we can construct a random binary search tree T_n on n nodes by taking a copy R of R_∞ and letting T_n consist of those nodes y of R with $h_n(y) \geq 1$. (See, e.g., Devroye [4].)

Fact 2. Let y be a node of R_∞ at depth i (i.e., at edge-distance i from the root). Then

$$nf(y) - i \leq h_n(y) \leq nf(y) .$$

Facts 1 and 2 basically allow us to obtain refined information regarding H_n merely by studying R_∞. The inequality in Fact 2 introduces a certain looseness; in fact, it will limit the accuracy of the results on H_n to be $O(\log \log n)$.

3. Lemmas regarding the gamma distribution. The sum S_n of n i.i.d. exponential random variables with mean 1 is gamma (n) distributed. Its density is given by

$$g(t) = \frac{t^{n-1}e^{-t}}{(n-1)!}, \qquad t > 0 .$$

LEMMA 1. *Let $\{t_n\}$ be a sequence of numbers such that $t_n \sim cn$ as $n \to \infty$ for some $c \in (0, 1)$. Then*

$$\mathbf{P}\{S_n < t_n\} \sim \frac{1}{1-c} \frac{e^{-t_n}(t_n)^n}{n!} .$$

Proof. By integration by parts,

$$\begin{aligned} \mathbf{P}\{S_n < t_n\} &= \int_0^{t_n} \frac{t^{n-1}e^{-t}}{(n-1)!}\, dt \\ &= e^{-t_n}\left(\frac{t_n^n}{n!} + \frac{t_n^{n+1}}{(n+1)!} + \frac{t_n^{n+2}}{(n+2)!} + \cdots\right) \\ &\sim \frac{1}{1-c}\frac{e^{-t_n}(t_n)^n}{n!} . \end{aligned}$$

□

LEMMA 2. *Let $t \in (0, 1)$ be a fixed constant. Then*

$$\frac{e^{-tn}(tn)^n}{n!} \leq \mathbf{P}\{S_n < tn\} \leq \frac{1}{1-t}\frac{e^{-tn}(tn)^n}{n!} .$$

Proof. The lower bound follows directly by integration by parts as in the proof of Lemma 1. For the upper bound, note that

$$\begin{aligned} \mathbf{P}\{S_n < tn\} &\leq e^{-tn}\left(\frac{(tn)^n}{n!} + \frac{(tn)^{n+1}}{(n+1)!} + \frac{(tn)^{n+2}}{(n+2)!} + \cdots\right) \\ &\leq \frac{e^{-tn}(tn)^n}{n!}\left(1 + \frac{tn}{n+1} + \left(\frac{tn}{n+1}\right)^2 + \cdots\right) \\ &\leq \frac{e^{-tn}(tn)^n}{n!}\left(\frac{1}{1-t}\right). \end{aligned}$$

□

LEMMA 3.

$$A \le \sqrt{n} 2^n \mathbf{P}\{S_n < n/\alpha\} \le B ,$$

where $A = e^{-1/12}/\sqrt{2\pi}$ *and* $B = \alpha/((\alpha - 1)\sqrt{2\pi})$.

Proof. From Lemma 2,

$$\frac{e^{-n/\alpha}(n/\alpha)^n}{n!} \le \mathbf{P}\{S_n < n/\alpha\} \le \frac{1}{1 - 1/\alpha} \frac{e^{-n/\alpha}(n/\alpha)^n}{n!} .$$

Use the fact that $n! = (n/e)^n \sqrt{2\pi n} e^{\theta/(12n)}$ for some $\theta \in (0, 1)$ and the definition of α. □

LEMMA 4. *There exists a universal constant* C *such that*

$$\mathbf{P}\{S_n \ge Cn\} \le 2^{-2n} .$$

$C = 5$ *will do.*

Proof. Take $C > 1$. By Chernoff's exponential bounding method (Chernoff [3]), for $t > 0$,

$$\mathbf{P}\{S_n \ge Cn\} \le \mathbf{E}\, e^{tS_n} e^{-tCn} = (1 - t)^{-n} e^{-tCn} = \left(Ce^{1-C}\right)^n ,$$

where we take $1 - t = 1/C$. For C large enough (e.g., $C \ge 5$), this is less than 4^{-n}. □

LEMMA 5. *Let* $E_1, E_2, \ldots, E_n$ *be i.i.d. random variables with a density, and let* a *be a fixed constant. Then*

$$\mathbf{P}\{E_1 < a, E_1 + E_2 < 2a, \ldots, E_1 + \cdots + E_n < na \mid E_1 + \cdots + E_n < na\} \ge \frac{1}{n} .$$

Proof. Define $F_i = E_i - a$ for all i. Define $E_r = E_{r-n}$, when $n < r \le 2n$. Then, by symmetry,

$$\begin{aligned}
&\mathbf{P}\{E_1 < a, E_1 + E_2 < 2a, \ldots, E_1 + \cdots + E_n < na \mid E_1 + \cdots + E_n < na\} \\
&\quad = \mathbf{P}\{F_1 < 0, F_1 + F_2 < 0, \ldots, F_1 + \cdots + F_n < 0 \mid F_1 + \cdots + F_n < 0\} \\
&\quad = \frac{1}{n} \sum_{i=1}^{n} \mathbf{P}\{F_i < 0, F_i + F_{i+1} < 0, \ldots, F_i + \cdots + F_{i+n-1} < 0 \mid F_1 + \cdots + F_n < 0\} \\
&\quad = \mathbf{P}\{F_S < 0, F_S + F_{S+1} < 0, \ldots, F_S + \cdots + F_{S+n-1} < 0 \mid F_1 + \cdots + F_n < 0\} ,
\end{aligned}$$

where S is independent of the E_i's and uniformly distributed on $\{1, \ldots, n\}$. Now, fix $E_1, \ldots, E_n$, and let $s \in \{1, \ldots, n\}$ be the (unique) value at which $\sum_{i>0, i<s} F_i$ is maximal. If $s = 1$, then $\sum_{i=1}^{j} F_i < 0$ for all $j > 0$. If $s > 1$, then, as $\sum_{i=1}^{n} F_i < 0$, we see that $\sum_{i=s}^{s+j} F_i = \sum_{i=1}^{s+j} F_i - \sum_{i=1}^{s-1} F_i < 0$ for all $j \ge 0$. Thus,

$$\begin{aligned}
&\mathbf{P}\{F_S < 0, F_S + F_{S+1} < 0, \ldots, F_S + \cdots + F_{S+n-1} < 0 \mid F_1 + \cdots + F_n < 0\} \\
&\quad \ge \mathbf{P}\{S = s\} = \frac{1}{n} .
\end{aligned}$$

□

4. Proof of the theorem.

LEMMA 6. *Consider positive integers* $n > k$. *Then*

$$\mathbf{P}\{H_n \ge k\} \ge \mathbf{P}\{\exists \text{ leaf } y \in R_k : f(y) \ge (k + 1)/n\} .$$

Proof. This follows immediately from Facts 1 and 2. □

LEMMA 7. *There exists a constant* $d > 0$ *such that for sufficiently large* j,

$$\mathbf{P}\left\{\exists \text{ leaf } y \in R_j : f(y) \ge (j + 1)/\exp(j/\alpha + d \log(j/\alpha))\right\} \ge 1 - \frac{1}{j^3} .$$

We may pick $d = \epsilon + 15/\log 2$ *for any small* $\epsilon > 0$.

Proof. The proof is contained in §5. □

LEMMA 8. *Let d be the constant of Lemma 7. Then, for sufficiently large n,*

$$\mathbf{P}\ \{H_n \geq \alpha \log n - d\alpha \log\log n - 1\} \geq 1 - \frac{1}{(\alpha \log n)^3}\ .$$

We may choose $d = \epsilon + 15/\log 2$ for any small $\epsilon > 0$.

Proof. The proof follows from Lemmas 6 and 7 by setting $j = k = \lfloor \alpha \log n - d\alpha \log\log n \rfloor$. □

LEMMA 9.

$$\mathbf{P}\ \{H_n \geq \lceil \alpha \log n + i \rceil\} \leq \left(\frac{2}{\alpha}\right)^i, \qquad i \geq 0\ .$$

Proof. See Devroye [4, p. 492]. □

Note that the theorem follows from Lemmas 8 and 9 without work.

5. Proof of Lemma 7.

LEMMA 10. *For every i with probability at least $1-2^{-i}$, every leaf of R_i has $f(y) \geq e^{-5i}$.*

Proof. The probability that, for some leaf y of R_i, we have $f(y) < e^{-5i}$ is at most 2^i times $\mathbf{P}\{S_i \geq 5i\}$, where S_i is gamma i distributed. By Lemma 4, this does not exceed $2^i/4^i = 2^{-i}$. □

LEMMA 11. *For sufficiently large k with probability at least $1/k^3$, there is a leaf y of R_k with $f(y) \geq e^{-k/\alpha}$.*

Lemma 11 will be proved in §6. If Lemma 11 is true, then we can proceed with the proof of Lemma 7 as follows: First note that we can obtain a copy of R_{i+k} by making each leaf of R_i a root of a copy of R_k, where all these trees are independently labelled. Define $k = \lfloor j - A \log j \rfloor$ and $i = \lceil A \log j \rceil$ so that $j = k + i$ with some constant A to be picked further on. Note first that for j large enough, if $A > \alpha$,

$$\frac{k}{\alpha} + 5i \leq \frac{j}{\alpha} + 5A \log\left(\frac{j}{\alpha}\right) - \log(j+1)\ .$$

Then,

$$\begin{aligned}
&\mathbf{P}\ \left\{\nexists \text{ leaf } y \in R_j \text{ with } f(y) \geq 1/\exp(j/\alpha + 5A\log(j/\alpha) - \log(j+1))\right\} \\
&\leq \mathbf{P}\ \left\{\exists \text{ leaf } y \in R_i \text{ with } f(y) < e^{-5i}\right\} \\
&\quad + \mathbf{P}\ \left\{\nexists \text{ leaf } y \in R_j \text{ with } f(y) \geq 1/\exp(j/\alpha + d\log(j/\alpha) - \log(j+1))\right. \\
&\qquad \left. \mid \forall \text{ leaf } y \in R_i : f(y) \geq e^{-5i}\right\} \\
&\leq 2^{-i} + \mathbf{P}\ \{\text{every copy of } R_k \text{ contains no leaf } y \text{ with } f(y) \geq 1/\exp(k/\alpha)\} \\
&\quad \text{(by Lemma 10)} \\
&\leq 2^{-i} + \left(1 - k^{-3}\right)^{2^i} \quad \text{(by Lemma 11)} \\
&\leq 2^{-i} + \exp\left(-2^i k^{-3}\right) \\
&\leq j^{-A\log 2} + \exp\left(-j^{A\log 2 - 3}\right) \\
&\leq j^{-3}
\end{aligned}$$

for j large enough, provided that $A \log 2 > 3$. This proves Lemma 7. We note that we can pick $d = 5A$, where $A = \epsilon + \max(\alpha, 3/\log 2)$ for any small $\epsilon > 0$.

6. Proof of Lemma 11. Let P be a path from the root to a leaf y of R_k. The condition $f(y) \geq 1/e^{k/\alpha}$ is equivalent to

$$\sum_{e \in P} (-\log L(e)) \leq \frac{|P|}{\alpha} .$$

We call a leaf y special if, in addition to the above condition, it satisfies

$$\sum_{e \in P'} (-\log L(e)) \leq \frac{|P'|}{\alpha}$$

for every subpath P' of P that originates at a terminal vertex y. Such subpaths are called terminal. Let $\mathcal{S}$ be the collection of special leaves of R_k. By Lemma 5, the expected number of special leaves is at least $1/k$ times $\mathbf{P}\{S_k < k/\alpha\}$ times 2^k. By Lemma 3,

$$\mathbf{E}|\mathcal{S}| \geq \frac{e^{-1/12}}{\sqrt{2\pi} k^{3/2}} .$$

Next, we consider the expected number of pairs of special leaves to be able to apply the second moment method. We fix a leaf z of R_k and count $|\mathcal{S}|$, given that $z \in \mathcal{S}$. To this end, let w be another leaf of R_k. Let P_z and P_w denote the paths from the root of R_k to z and w, respectively. Then P_z and P_w have an initial common subsequence, i.e., the join $P_z \cap P_w$. Let $e_1, e_2, \ldots, e_k$ be the edges on the path from the root to z and define $Q_i = \{e_1, \ldots, e_i\}$. For any j, the number of leaves of R_k whose join with P_z is Q_j is 2^{k-j}. Furthermore, the probability that a leaf $w \in R_k$ is a special leaf, given that $z \in \mathcal{S}$ and $P_z \cap P_w = Q_j$, is bounded above by the probability that for the terminal path $P' \subseteq P_w - Q_j$ with $|P'| = \max(0, k - j - 1)$, we have

$$\sum_{e \in P'} (-\log L(e)) \leq \frac{|P'|}{\alpha} .$$

Note that P' contains one edge less than $P_w - Q_j$. Later, this allows us to work out a conditional probability, given $z \in \mathcal{S}$, without much trouble. By Lemma 3, the probability of the event mentioned above is at most

$$\frac{\alpha}{(\alpha - 1)\sqrt{2\pi(k - j - 1)} 2^{k-j-1}} .$$

Thus,

$$\begin{aligned}
&\mathbf{E}\left\{|\{w \in \mathcal{S} : P_w \cap P_z = Q_j\}| \mid z \in \mathcal{S}\right\} \\
&\leq \frac{\alpha 2^{k-j}}{(\alpha - 1)\sqrt{2\pi(k - j - 1)} 2^{k-j-1}} \\
&= \frac{2\alpha}{(\alpha - 1)\sqrt{2\pi(k - j - 1)}} \\
&\leq 2,
\end{aligned}$$

when $k-j \geq 2$. The previous expected value is bounded by 2 when $k-j \in \{0, 1\}$. Therefore,

$$\begin{aligned}\mathbf{E}\ \{|\mathcal{S}| \mid z \in \mathcal{S}\} &= \sum_{j=0}^{k} \mathbf{E}\ \left\{|\{w \in \mathcal{S} : P_w \cap P_z = Q_j\}| \mid z \in \mathcal{S}\right\} \\ &\leq \sum_{j=0}^{k} 2 = 2k+2\,.\end{aligned}$$

Hence, by the second moment method,

$$\begin{aligned}\mathbf{P}\{|\mathcal{S}| \geq 1\} &\geq \frac{\mathbf{E}|\mathcal{S}|}{1+\sup_{z \text{ leaf of } R_k} \mathbf{E}\{|\mathcal{S}||z \in \mathcal{S}\}} \\ &\geq \frac{\mathbf{E}\,|\mathcal{S}|}{2k+3} \\ &\geq \frac{e^{-1/12}}{\sqrt{2\pi}(2k+3)k^{3/2}} \\ &\geq \frac{1}{k^3}\end{aligned}$$

for all k large enough. This concludes the proof of Lemma 11.

Acknowledgments. The authors thank Colin McDiarmid and an anonymous referee for helpful comments.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1975.
[2] ———, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.
[3] H. CHERNOFF, *A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations*, Ann. Math. Statist., 23 (1952), pp. 493-507.
[4] L. DEVROYE, *A note on the height of binary search trees*, J. Assoc. Comput. Mach., 33 (1986), pp. 489-498.
[5] ———, *Branching processes in the analysis of the heights of trees*, Acta Inform., 24 (1987), pp. 277-298.
[6] ———, *On the height of random m-ary search trees*, Random Structures Algorithms, 1 (1990), pp. 191-203.
[7] D. E. KNUTH, *The Art of Computer Programming, Vol.* 1: *Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1973.
[8] ———, *The Art of Computer Programming, Vol.* 3: *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.
[9] H. M. MAHMOUD, *Evolution of Random Search Trees*, John Wiley, New York, 1992.
[10] H. MAHMOUD AND B. PITTEL, *On the most probable shape of a search tree grown from a random permutation*, SIAM J. Algebraic Discrete Meth., 5 (1984), pp. 69-81.
[11] E. M. PALMER, *Graphical Evolution*, John Wiley, New York, 1985.
[12] B. PITTEL, *On growing random binary trees*, J. Math. Anal. Appl., 103 (1984), pp. 461-480.
[13] ———, *Note on the heights of random recursive trees and random m-ary search trees*, Tech. report, Department of Mathematics, Ohio State University, 1992.
[14] R. PYKE, *Spacings*, Roy. Statist. Soc. Ser. B, 7 (1965), pp. 395-445.
[15] J. M. ROBSON, *The height of binary search trees*, Austral. Comput. J., 11 (1979), pp. 151-153.
[16] ———, *The asymptotic behaviour of the height of binary search trees*, Austral. Comput. Sci. Comm., Queensland Univ. Tech., Brisbane, 1982, p. 88.

SIAM J. COMPUT.
Vol. 24, No. 6, pp. 1163-1169, December 1995

CONSTRUCTING HUFFMAN TREES IN PARALLEL*

LAWRENCE L. LARMORE† AND TERESA M. PRZYTYCKA‡

Abstract. We present a parallel algorithm for the Huffman coding problem. We reduce the Huffman coding problem to the *concave least weight subsequence* (CLWS) problem and give a parallel algorithm that solves the latter problem in $O(\sqrt{n}\log n)$ time with n processors on a concurrent read exclusive write parameter random-access machine (CREW PRAM). This leads to the first sublinear-time $o(n^2)$-total-work parallel algorithm for Huffman coding. This reduction of the Huffman coding problem to the CLWS problem also yields an alternative $O(n \log n)$-time (or linear-time, for a sorted input sequence) algorithm for Huffman coding.

Key words. Huffman coding, parallel algorithms

AMS subject classification. 68P20

1. Introduction.

Throughout this paper, a *tree* is a regular binary tree (i.e., a binary tree in which each internal node has two children). The *level* of a node in a tree is its distance from the root. The problem of constructing a *Huffman tree*, given a sequence of n nonnegative real numbers, $x_1, x_2, \ldots, x_n$, is to construct a tree with n leaves, where the leaves of the tree are in one-to-one correspondence[1] with elements of the sequence to minimize the following cost function:

$$c(T) = \sum_{i=1}^{n} x_i \ell_i, \tag{1}$$

where ℓ_i is the level of the leaf corresponding to x_i. The value x_i associated with a leaf v is called the *weight* of v.

The Huffman tree problem, also called the *Huffman coding* problem, can be solved in $O(n \log n)$ sequential time [9], or linear time if the input sequence is sorted. Despite substantial effort, however, no good parallel algorithm is known for the problem. Currently, the best $\mathcal{NC}$ algorithm for the Huffman coding problem takes $O(\log^2 n)$ time with roughly n^2 concurrent read exclusive write parameter random-access machine (CREW PRAM) processors [3]. An approximate solution for the Huffman coding problem can be computed in $O(\log n \log^* n)$ time using a linear number of CREW processors [10]. In this paper we present the first sublinear-time, $o(n^2)$-work parallel algorithm for the Huffman coding problem. At the heart of our algorithm is the reduction of the Huffman coding problem to the *concave least weight subsequence* (CLWS) problem. This reduction leads to a new linear-time (if the input sequence of weights is sorted) sequential algorithm for the Huffman coding problem.

The CLWS problem has a number of applications, including paragraph breaking.[2] A good parallel algorithm for this problem is thus interesting in its own right. Hirschberg and Larmore [7] define the *least weight subsequence* (LWS) problem as follows: Given an integer n and a real-valued *weight* function $w(i, j)$ defined for integers $0 \leq i < j \leq n$, find a sequence of

*Received by the editors June 24, 1992; accepted for publication (in revised form) May 23, 1994.

†Department of Computer Science, University of Nevada, Las Vegas, Nevada 89154-4019 (larmore@earl.cs.unlv.edu). This research was done while the author was at the Department of Computer Science, University of California, Riverside. The research of this author was supported by National Science Foundation grant CCR9112067.

‡Department of Mathematics and Computer Science, Odense University, DK-5230 Odense M, Denmark (przytyck@imada.ou.dk). A part of this work was done while this author was visiting the University of California, Riverside.

[1]This correspondence need not preserve order.

[2]The *paragraph breaking* problem involves finding the best way of breaking a paragraph of text into lines, a problem that must be handled by text processing software such as TEX.

integers $\overline{\alpha} = (0 = \alpha_0 < \alpha_1 < \cdots < \alpha_{k-1} < \alpha_k = n)$ such that $w(\overline{\alpha}) = \sum_{i=0}^{k-1} w(\alpha_i, \alpha_{i+1})$ is minimized. Thus, the LWS problem reduces trivially to the minimum path problem on a weighted directed acyclic graph. The *single source* LWS problem involves finding such a minimal sequence $0 = \alpha_0 < \alpha_1 < \cdots < \alpha_{k-1} < \alpha_k = m$ for all $m \leq n$. The weight function is said to be *concave* if, for all $0 \leq i_0 \leq i_1 < j_0 \leq j_1 \leq n$,

$$w(i_0, j_0) + w(i_1, j_1) \leq w(i_0, j_1) + w(i_1, j_0). \tag{2}$$

The inequality (2) is also called the *quadrangle inequality* [17].

The LWS problem, with the restriction that the weight function is concave, is called the CLWS problem. Hirschberg and Larmore [7] show that the LWS problem can be solved in $O(n^2)$ sequential time, while the CLWS problem can be solved in $O(n \log n)$ time. Galil and Giancarlo [5] give an $O(n \log n)$-time bound for the *convex least weight subsequence* problem (i.e., when the weight function satisfies the reverse of inequality (2)). Wilber [16] gives an $O(n)$-time algorithm for the CLWS problem. Linear-time algorithms for this problem are also given by Klawe [12], Larmore and Schieber [15], and Galil and Park [6]. The best-known sequential algorithm for the convex case is by Klawe and Kleitman [13]. Their algorithm requires $O(n\alpha(n))$ time, where α is the inverse of the Ackerman function. All of these algorithms actually solve the single source problems.

In the parallel setting, the CLWS problem seems to be more difficult than the corresponding convex problem. Lam and Chan [14] present an $O(\log^2 n \log\log n)$-time, $n/\log\log n$-processor CREW PRAM algorithm to solve the convex problem. On the other hand, the best current $\mathcal{NC}$ algorithm for the CLWS problem uses concave matrix multiplication techniques [1]–[3] and requires $O(\log^2 n)$ time with $n^2/\log^2 n$ processors. In this paper, we present an $O(\sqrt{n} \log n)$-time n-processor CREW PRAM algorithm to solve the CLWS problem.

This paper is organized as follows. In §2, we give a parallel algorithm for the CLWS problem. In §3, we reduce the Huffman coding problem to the CLWS problem. We conclude with some open questions.

2. The parallel single source CLWS algorithm. Consider an instance of the CLWS problem over $[0, n]$ defined by a weight function w, and let $\mathrm{lws}(i, j)$ denote the LWS problem on $[i, j]$ defined by restricting w to pairs within that interval. Let $W(i, j)$ be the weight of an optimal solution to $\mathrm{lws}(i, j)$. For convenience, we let $W(i, i) = 0$ for all i.
Define

$$f(j) = W(0, j)$$

and

$$\mathrm{pred}(j) = \begin{cases} \text{undefined} & \text{for } j = 0, \\ \min\{i \mid i < j,\ f(j) = f(i) + w(i, j)\} & \text{for } 0 < j \leq n, \end{cases}$$

$$\mathcal{F}(j) = (f(j), \mathrm{pred}(j)).$$

Thus $f(j)$ is equal to the weight of an LWS for the interval $[0, j]$ and $\mathrm{pred}(j)$ is equal to the index of the second-to-last element of such a subsequence. (In case of ties, we let $\mathrm{pred}(j)$ be as small as possible.) To solve the LWS problem it suffices to compute the pair $\mathcal{F}(j)$ for all $j \in [0, n]$.

Given an interval $I = [i_1, i_2] \subseteq [0, n]$ we can also consider a restricted version of the LWS problem in which we require that, for all $j > i_2$, $\mathrm{pred}(j) \in I$ (i.e., for any $j > i_2$ the second-to-last element of the solution to $\mathrm{lws}(0, j)$ belongs to I). To solve this problem it

suffices to compute the pair $\mathcal{F}^I(j)$ for all $0 < j \le n$, where

$$f^I(j) = \begin{cases} W(0, j) & \text{for } j \le i_2, \\ \min_{i \in I}(f(i) + w(i, j)) & \text{for } i_2 < j \le n, \end{cases}$$

$$\text{pred}^I(j) = \begin{cases} \text{undefined} & \text{for } j = 0, \\ \min\{i \mid i < j, f(j) = f(i) + w(i, j)\} & \text{for } 0 < j \le i_2, \\ \min\{i \mid i \in I, i < j, f^I(j) = f(i) + w(i, j)\} & \text{for } i_2 < j \le n, \end{cases}$$

$$\mathcal{F}^I(j) = (f^I(j), \text{pred}^I(j)).$$

Computing $\mathcal{F}^{[0,km]}$, given $\mathcal{F}^{[0,(k-1)m]}$ and $\mathcal{F}^{[(k-1)m+1,km]}$, is relatively easy and is based on the following fact.

FACT 2.1. *Let* $I = [0, i]$. *Then* $f(k) = \min\{j \in [i+1, k] \mid f^I(j) + W(j, k)\}$ *for any* $k > i$.

Proof. Let j be the first element of the solution to lws$(0, k)$ such that $j > i$. It is possible that $j = k$. Then $f(k) = f(j) + W(j, k) = f^I(j) + W(j, k)$. By the minimality of the solution, $f(k) \le f^I(j') + W(j', k)$ for any $j' \le k$. □

2.1. The dominance partition. Throughout this subsection, fix $I = [i_1, i_2]$. For any $i \in I$, let $D^I(i) = \{j > i_2 \mid \text{pred}^I(j) = i\}$. We shall see that if $D^I(i)$ is not empty, it must be an interval, which we call the interval *dominated* by i.

For any $r < s$, let boundary(r, s) be the maximum $j > s$ for which $f^r(j) \le f^s(j)$. If there is no such j we use the default value boundary$(r, s) = s$.

LEMMA 2.2. *Let* $r < s$. *Then* $f^s(k) - f^r(k)$ *is a monotone decreasing function of* k.

Proof. Let $s < k < k'$. Then

$$f^s(k') - f^r(k') - f^s(k) + f^r(k) = w(s, k') - w(r, k') - w(s, k) + w(r, k) \le 0$$

by (2). □

LEMMA 2.3. *For any* $r < s$, *given the values of the function* $f(k)$ *for all* $k \le s$, *the value* boundary(r, s) *can be computed in* $O(\log n)$ *sequential time.*

Proof. The correct value of k = boundary(r, s) is the maximum k for which $f^s(k) - f^r(k) \ge 0$, which can be found by binary search by Lemma 2.2. □

LEMMA 2.4. *Let* $i \in I$. *Then* $D^I(i)$ *is either empty or an interval.*

Proof. By Lemma 2.2,

$$D^I(i) = \bigcap_{i_1 \le r < i} [\text{boundary}(r, i) + 1, n] \cap \bigcap_{i < s \le i_2} [i_2 + 1, \text{boundary}(i, s)].$$

The intersection of any number of intervals is either empty or an interval. □

LEMMA 2.5. *Let* $i', i'' \in I$, *where* $i' < i''$. *Then* $D^I(i')$ *lies entirely to the left of* $D^I(i'')$ *if both are nonempty.*

Proof. $D^I(i')$ lies entirely in the interval $[i_2, \text{boundary}(i', i'')]$, while $D^I(i'')$ lies entirely to the right of boundary(i', i''). □

LEMMA 2.6. *Let* $I = [i, i + m]$. *Given* $\mathcal{F}^{[0,i-1]}$, $\mathcal{F}^I$ *can be computed in* $O(\log n + m^2 \log n/p + n \log n/p)$ *time with* p *CREW PRAM processors.*

Proof. Let $\text{right}^I(i) = \min\{\text{boundary}(i, s) \mid i < s \le i_2\}$ and $\text{left}^I(i) = \max\{\text{boundary}(r, i) \mid i_2 \le r < i\}$. If $\text{left}^I(i) > \text{right}^I(i)$, then $D^I(i) = \emptyset$. Otherwise, $D^I(i) = [\text{left}^I(i), \text{right}^I(i)]$. We can compute $\text{right}^I(i)$ for all $i \in I$ using m^2 processors in

$O(\log n)$ time as follows. Each processor computes boundary(i, j) for one pair $i < j$ in I using binary search (see Lemma 2.3). All values of $\text{right}^I(i) = \min\{\text{boundary}(i, j) \mid i < j \leq i_2\}$ can be computed in $O(\log m)$ time with $m^2/\log m$ processors. Values of left^I are computed similarly.

All values of pred^I can then be computed in $O(\log n)$ time with $n/\log n$ processors. If $k = \text{left}^I(i) \leq \text{right}^I(i)$, initialize $\text{pred}^I(k) := i$, otherwise initialize $\text{pred}^I(k) := 0$. Using a prefix-sum technique, all correct values of pred^I are then computed. Finally, $f^I(k)$ is computed using pred^I in $O(1)$ time using n processors.

The result for p processors follows by Brent's principle. □

2.2. The algorithm. Fix a parameter $m \leq n$.

ALGORITHM CLWS

1. *preprocessing:*
for all i, j such that $i < j$ and $j - i \leq m$ **do** solve lws(i, j);
2. *initialization:*
for all $0 \leq j \leq n$ **do** $f^{[0,0]}(j) = 0$,
3. *iteratively compute* $\mathcal{F}^{[0,im]}$*:*
for $i := 1$ **to** n/m **do**
    3.1. **for all** $k \in [(i-1)m + 1, im]$ **do** compute $\mathcal{F}^{[0,im]}(k)$ $(= \mathcal{F}(k))$
    3.2. **if** $i < n/m$ **then**
        3.2.1 compute $\mathcal{F}^{[(i-1)m+1,im]}$;
        3.2.2 compute $\mathcal{F}^{[0,im]}$.

THEOREM 2.7. *Algorithm CLWS can be implemented to run in* $O(n \log n/m + n^2/mp + nm \log n/p)$ *time with* p *CREW PRAM processors.*

Proof. The implementation of step 2 is straightforward. We need to show the implementation of steps 1 and 3 and prove correctness.

STEP 1.

Implementation. Assign one processor to each element of the sequence. Compute the solution to lws(i, j) for all pairs $0 \leq i < j \leq n$, such that $j \leq i + m$, using a linear-time algorithm for one instance of size m of the single source CLWS problem for each i.

Correctness. The correctness is immediate.

Complexity. The complexity is $O(m)$ time with n processors.

STEP 3.1.

Implementation. Let $f^{[0,im]}(k) = \min_{j \in [(i-1)m, im], j \leq k} \left\{ f^{[0,(i-1)m]}(j) + W(j, k) \right\}$ for all k in the interval $[(i-1)m + 1, im]$.

Correctness. The correctness is proven by Fact 2.1.

Complexity. Assign one processor to each pair $j, k \in I$, $j < k$, and compute for each such pair the value $f^{[0,(i-1)m]}(j) + W(j, k)$. This requires $O(1)$ time using m^2 processors. Over all iterations, $O(n/m)$ time is needed. Then we compute the corresponding minima in $O(\log m)$ time using $m^2/\log m$ processors. Over all iterations of the main loop, this step requires $O(n \log m/m)$ total time with $m^2/\log m$ processors.

STEP 3.2. By Lemma 2.6, this step takes $O(\log n)$ time with m^2 processors at each iteration. Over all iterations of the main loop, this step requires $O(n \log n/m)$ total time with m^2 processors.

STEP 3.3.

Implementation. For all $j \in [im + 1, n]$, let

$$f^{[0,im]}(j) = \min \left\{ f^{[0,(i-1)m]}(j),\ f^{[(i-1)m+1,im]}(j) \right\}.$$

Correctness. The correctness is immediate by the definition of f^I, since $[0, im] = [0, (i-1)m] \cup [(i-1)m+1, im]$.

Complexity. The complexity is $O(1)$ time with n processors.

The statement of the lemma follows from Brent's principle. □

3. Reduction of the Huffman tree problem to CLWS. A regular binary tree can be uniquely described by listing the levels of its leaves in left to right order. If T is a regular binary tree and ℓ_i is the level of the ith leaf of T, where the leaves are indexed from left to right, we call $(\ell_1, \ell_2, \ldots, \ell_n)$ the *leaf pattern* of T. A tree is said to be *left justified* [3] if its leaf pattern is nonincreasing.

LEMMA 3.1 [8]. *Given an arbitrary input sequence $\bar{x}$, there is a Huffman tree for $\bar{x}$ that is left justified.*

For the remainder of this section we assume that $n \geq 2$, because the case $n = 1$ is trivial. By Lemma 3.1, sorting reduces our initial problem to that of constructing a left-justified tree which minimizes the cost function c. We show that the latter problem can be reduced to the CLWS problem. A left-justified tree T of height k can be uniquely described by listing the numbers of internal nodes on each level of the tree. Thus T can be described by a monotone sequence of integers $0 = \alpha_0, \alpha_1, \ldots, \alpha_{k-1}, \alpha_k = n-1$, where, for any $j > 0$, $\alpha_j - \alpha_{j-1}$ is equal to the number of internal nodes on level $k-j$. We call such a sequence *the level sequence* of the tree T. Observe that not every monotone sequence from 0 to $n-1$ is a level sequence for a left-justified tree. For example, if $n = 4$, (0,1,2,3) and (0,2,3) are level sequences for left-justified trees, while (0,3) and (0,1,3) are not.

A simple but important property of a level sequence is given in the following lemma.

LEMMA 3.2. *Let T be a left-justified tree with level sequence $0 = \alpha_0, \alpha_1, \ldots, \alpha_{k-1}, \alpha_k = n-1$. Then, for any $0 \leq i < k$, T has precisely $2\alpha_{i+1} - \alpha_i$ leaves at levels $k-i$ and below.*

Proof. Let β_i be the number of leaves of T at levels $k-i, k-i+1, \ldots k$, i.e., at levels $k-i$ and below. By the definition of a level sequence, α_i is the number of internal nodes of T at level $k-i$ and below. For any forest consisting of regular binary trees, the number of leaves is the sum of the number of internal nodes and the number of roots. Let F be the forest consisting of all internal nodes of T at level $k-i-1$ and all their descendants. F has α_{i+1} internal nodes, $\alpha_{i+1} - \alpha_i$ roots, and β_i leaves. Therefore $\beta_i = \alpha_{i+1} + (\alpha_{i+1} - \alpha_i) = 2\alpha_{i+1} - \alpha_i$. □

As an immediate consequence of Lemma 3.2 we obtain the following lemma.

LEMMA 3.3. *A left-justified tree T can be computed from its level sequence $\bar{\alpha}$ in $O(\log n)$ time with $n/\log n$ exlusive read exclusive write* (EREW) *processors.*

Proof. Using Lemma 3.2, compute the number of leaves at each level. Then, using a prefix-sum computation, for each i for $1 \leq i \leq n$, compute the level of the ith leaf in T. Given the sequence of its leaf levels, T can be constructed in $O(\log n)$ time with $n/\log n$ processors [11]. □

The main result of this section is stated in the following theorem.

THEOREM 3.4. *Let $s_1, s_2, \ldots, s_n$ be the sequence of prefix sums for a nondecreasing sequence $\bar{x} = x_1, x_2, \ldots, x_n$, i.e., $s_i = \sum_{j=1}^{i} x_j$, and let $w(i, j)$, be the weight function defined as follows:*

$$w(i,j) = \begin{cases} s_{2j-i} & \text{if } 2j - i \leq n, \\ \infty & \text{otherwise;} \end{cases}$$

then the LWS problem defined by w is concave and the solution to this problem is equal to the level sequence of the left-justified Huffman tree for $\bar{x}$.

Proof. We define a strictly monotone sequence of integers $\overline{\alpha} = (0 = \alpha_0, \ldots \alpha_k = n-1)$, for $n > 0$, as *regular* if $2\alpha_i - \alpha_{i-1} \leq 2\alpha_{i+1} - \alpha_i \leq n$ for all $0 < i < k$. Note that if $\overline{\alpha}$ is regular, $\alpha_{k-1} = n-2$.

We make the following three claims.

CLAIM A. *If T is any left-justified tree and $\overline{\alpha}$ is the level sequence for T, then $\overline{\alpha}$ is regular. Furthermore, $c(T) = w(\overline{\alpha})$.*

CLAIM B. *If $\overline{\alpha}$ is regular, then it is the level sequence of some left-justified tree.*

CLAIM C. *If $\overline{\alpha}$ is a monotone sequence from 0 to $n-1$ which is not regular, then there is some regular monotone sequence from 0 to $n-1$ whose weight is no greater than that of $\overline{\alpha}$.*

Proof of Claim A. Let β_i be the number of leaves at levels $k-i$ and below. Trivially, $\beta_i \leq \beta_{i+1}$. Since $\beta_i = 2\alpha_{i+1} - \alpha_i$, by Lemma 3.2 we have that $\overline{\alpha}$ is regular. There are β_0 leaves of T at level k and $\beta_i - \beta_{i-1}$ leaves at level $k-i$ for $0 < i \leq k$. Since T is left justified, the total weight of the leaves at level $k-i$ is s_{β_0} if $i = 0$ and $s_{\beta_i} - s_{\beta_{i-1}}$ if $i > 0$. Thus

$$\begin{aligned} c(T) &= k s_{\beta_0} + \sum_{i=1}^{k} (k-i)(s_{\beta_i} - s_{\beta_{i-1}}) \\ &= \sum_{i=0}^{k-1} s_{\beta_i} = \sum_{i=0}^{k-1} w(\alpha_i, \alpha_{i+1}) = w(\overline{\alpha}). \end{aligned}$$

Proof of Claim B. Let $\beta_k = n$ and $\beta_i = 2\alpha_{i+1} - \alpha_i$ for $0 \leq i < k$. Since $\overline{\alpha}$ is regular, $\beta_i \leq \beta_{i+1}$ for all $0 \leq i < k$.

We construct T in a bottom-up fashion. Let F^1 be the forest obtained by combining the first β_0 leaves in pairs. For $i > 0$, let F^i be the forest obtained by combining the roots of F^{i-1} and the next $\beta_i - \beta_{i-1}$ leaves in pairs. By induction, F^i has $\alpha_i - \alpha_{i-1}$ roots, since $\beta_0 = 2\alpha_1 = 2(\alpha_1 - \alpha_0)$ and $\alpha_i - \alpha_{i-1} + \beta_i - \beta_{i-1} = 2(\alpha_{i+1} - \alpha_i)$ for $i > 0$.

Finally, let $T = F^k$. T is a tree, since $\alpha_k - \alpha_{k-1} = 1$. By the construction, $\overline{\alpha}$ is its level sequence.

Proof of Claim C. Define $\Sigma\overline{\alpha} = \sum_{i=0}^{k} \alpha_i$. Suppose $\overline{\alpha}$ is not regular. Choose $0 < i < k$, for which $2\alpha_i - \alpha_{i-1} = p > 2\alpha_{i+1} - \alpha_i = q$. Let $\overline{\alpha}' = (\alpha_0 \ldots \alpha_{i-1}, \alpha_i - 1, \alpha_{i+1}, \ldots \alpha_k)$. Then $w(\overline{\alpha}) - w(\overline{\alpha}') = s_p + s_q - s_{p-2} - s_{q+1} = x_p + x_{p+1} - x_{q+1} \geq 0$ and $\Sigma\overline{\alpha}' = \Sigma\overline{\alpha} - 1$. We repeat this construction as many times as necessary until a regular sequence is obtained. Since $\Sigma\overline{\alpha}$ is decremented at each step, the process must converge.

We now return to the proof of Theorem 3.4. Let $\overline{\alpha} = (0 = \alpha_0, \ldots \alpha_k = n-1)$ be an optimal solution to the LWS. By Claim C, we can assume that $\overline{\alpha}$ is regular. By Claim B, there is a left-justified tree T whose cost is $w(\overline{\alpha})$. By Claim A, T must be optimal.

We remark that a regular optimal solution can be guaranteed by adopting the rule that, in case of ties, pred(k) is chosen as small as possible. This is a rule which our parallel algorithm enforces. This finishes the proof of Theorem 3.4. □

COROLLARY 3.5. *For any value of the parameter m in the interval $[1, n]$, the Huffman coding problem can be solved in $O(n \log n/m + n^2/mp + nm \log n/p)$ time with p CREW PRAM processors.*

Proof. The proof is immediate by Theorems 2.7 and 3.4 and Lemma 3.3. □

COROLLARY 3.6. *The Huffman coding problem can be solved in $O(\sqrt{n} \log n)$ time with n CREW PRAM processors.*

Proof. Let $p = n$ and $m = \sqrt{n}$ and apply Corollary 3.5. □

4. Open questions. In this paper we present a parallel algorithm for the Huffman coding program that performs $O(n^{3/2} \log n)$ work. The algorithm requires $O(n^{1/2} \log n)$ time. Czumaj [4] presents a parallel CLWS algorithm that runs in $O(n^{1/2} \log^{3/2} n)$ time with $n/\log n$

processors, thus reducing the total work by a logarithmic factor. A challenging open question is whether there exists an $\mathcal{NC}$ algorithm that solves the Huffman coding problem with $o(n^2)$ total work.

Acknowledgments. We thank an anonymous referee for a large number of helpful comments.

REFERENCES

[1] A. Apostolico, M. J. Atallah, L. L. Larmore, and H. S. McFaddin, *Efficient parallel algorithms for string editing and related problems*, Proc. 26th Annual Allerton Conference on Communication, Control, and Computing, Monticello, IL, 1988, pp. 253–263; Tech. report CSD-TR-724, Purdue University, 1988; reprinted in SIAM J. Comput., 19 (1990), pp. 968–988.

[2] A. Aggarwal and J. Park, *Notes on searching in multidimensional monotone arrays*, Proc. 29th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1988, pp. 497–513.

[3] M. J. Atallah, S. R. Kosaraju, L. L. Larmore, G. L. Miller, and S.-H. Teng, *Constructing trees in parallel*, Proc. 1st Symposium on Parallel Algorithms and Architectures, ACM Press, 1989, pp. 499–533.

[4] A. Czumaj, *Parallel algorithm for the matrix chain product problem*, 1992, preprint.

[5] Z. Galil and R. Giancarlo, *Data structures and algorithms for approximate string matching*, J. Complexity, 1988, pp. 33–72.

[6] Z. Galil and J. Park, *A linear time algorithm for concave one-dimensional dynamic programming*, Inform. Process. Lett., 10 (1990), pp. 309–311.

[7] D. S. Hirschberg and L. L. Larmore, *The least weight subsequence problem*, Proc. 26th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1985, pp. 137–143; SIAM J. Comput., 16 (1987), pp. 628–638.

[8] T. C. Hu and K. C. Tan, *Path length of binary search trees*, SIAM J. Appl. Math., 22 (1972), pp. 225–234.

[9] D. A. Huffman, *A method for the constructing of minimum redundancy codes*, Proc. Institute of Radio Engineers, 40 (1952), pp. 1098–1101.

[10] D. G. Kirkpatrick and T. M. Przytycka, *Parallel construction of binary trees with almost optimal weighted path length*, Proc. 2nd Symposium on Parallel Algorithms and Architectures, 1990, pp. 234–243.

[11] ———, *An optimal parallel minimax tree algorithm*, Proc. 2nd IEEE Symposium on Parallel and Distributed Processing, IEEE Computer Society Press, 1990, pp. 293–300.

[12] M. M. Klawe, *A simple linear time algorithm for concave one-dimensional dynamic programming*, Tech. report TR 89-16, Department of Computer Science, University of British Columbia, 1989.

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

[14] T-W. Lam and K. F. Chan, *Finding least-weight subsequences with fewer processors*, Algorithmica, 9 (1993), pp. 615–628.

[15] L. L. Larmore and B. Schieber, *On-line dynamic programming with applications to the prediction of RNA secondary structure*, Proc. 1st ACM–SIAM Symposium on Discrete Algorithms, San Francisco, CA, 1990, pp. 503–512; J. Algorithms, 12 (1991), pp. 490–515.

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

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

SIAM J. COMPUT.
Vol. 24, No. 6, pp. 1170–1189, December 1995

004

WEAKLY HARD PROBLEMS*

JACK H. LUTZ†

Abstract. A weak completeness phenomenon is investigated in the complexity class $\mathrm{E} = \mathrm{DTIME}(2^{\mathrm{linear}})$. According to standard terminology, a language H is $\leq_m^{\mathrm{P}}$-*hard* for E if the set $\mathrm{P}_m(H)$, consisting of all languages $A \leq_m^{\mathrm{P}} H$, contains the entire class E. A language C is $\leq_m^{\mathrm{P}}$-*complete* for E if it is $\leq_m^{\mathrm{P}}$-hard for E and an element of E. Generalizing this, a language H is *weakly* $\leq_m^{\mathrm{P}}$-*hard* for E if the set $\mathrm{P}_m(H)$ does not have measure 0 in E. A language C is *weakly* $\leq_m^{\mathrm{P}}$-*complete* for E if it is weakly $\leq_m^{\mathrm{P}}$-hard for E and an element of E.

The main result of this paper is the construction of a language that is weakly $\leq_m^{\mathrm{P}}$-complete, but not $\leq_m^{\mathrm{P}}$-complete, for E. The existence of such languages implies that previously known strong lower bounds on the complexity of weakly $\leq_m^{\mathrm{P}}$-hard problems for E (given by work of Lutz, Mayordomo, and Juedes) are indeed more general than the corresponding bounds for $\leq_m^{\mathrm{P}}$-hard problems for E.

The proof of this result introduces a new diagonalization method called *martingale diagonalization*. Using this method, one simultaneously develops an infinite family of polynomial time computable martingales (betting strategies) and a corresponding family of languages that defeat these martingales (prevent them from winning too much money) while also pursuing another agenda. Martingale diagonalization may be useful for a variety of applications.

Key words. complete problems, complexity classes, computational complexity, resource-bounded measure, weak completeness

AMS subject classification. 68Q15

1. Introduction.

In practice to date, proving that a decision problem (i.e., language) $H \subseteq \{0, 1\}^*$ is computationally intractable usually amounts to proving that every member of the complexity class $\mathrm{E} = \mathrm{DTIME}(2^{\mathrm{linear}})$—or some larger class—is efficiently reducible to H. (See [25] for a survey of such arguments.) For example, some problems involving the existence of winning strategies for certain two-person combinatorial games are known to be intractable because they are polynomial time many-one hard (in fact, logarithmic space many-one complete) for E [24].

Briefly, a language H is *polynomial time many-one hard* (abbreviated $\leq_m^{\mathrm{P}}$-*hard*) for E if every language $A \in \mathrm{E}$ is polynomial time many-one reducible to H (abbreviated $A \leq_m^{\mathrm{P}} H$). A language C is $\leq_m^{\mathrm{P}}$-*complete* for E if $C \in \mathrm{E}$ and C is $\leq_m^{\mathrm{P}}$-hard for E.

A language H that is $\leq_m^{\mathrm{P}}$-hard for E is clearly intractable in the sense that $H \notin \mathrm{P}$, i.e., H is not decidable in polynomial time. This is because a well-known diagonalization argument [3] shows that there is a language $B \in \mathrm{E} - \mathrm{P}$. Since $B \in \mathrm{E}$, it must be the case that $B \leq_m^{\mathrm{P}} H$. Since $B \notin \mathrm{P}$, it follows that $H \notin \mathrm{P}$.

In fact, languages that are $\leq_m^{\mathrm{P}}$-hard for E are known to have much stronger intractability properties. Three examples follow:

(A) Meyer [15] has shown that every $\leq_m^{\mathrm{P}}$-hard language H for E is *dense*. This means that there is a real number $\varepsilon > 0$ such that, for all sufficiently large n, H contains at least 2^{n^ε} strings $x \in \{0, 1\}^{\leq n}$.

(B) Schöning [23] and Huynh [6] have shown that every $\leq_m^{\mathrm{P}}$-hard language H for E is hard to approximate in the sense that, for every language $A \in \mathrm{P}$, the symmetric difference $A \triangle H$ is dense. (Note that this immediately implies result (A) above.)

(C) Orponen and Schöning [16] have shown that every $\leq_m^{\mathrm{P}}$-hard language H for E has a dense polynomial complexity core K. This condition, defined precisely in §2, means, roughly, that K is dense and every Turing machine that is consistent with H performs

*Received by the editors May 24, 1993; accepted for publication (in revised form) May 31, 1994. This research was supported in part by National Science Foundation grant CCR-9157382 with matching funds from Rockwell International and Microware Systems Corporation.

†Department of Computer Science, Iowa State University, Ames, Iowa 50011.

badly (either by running for more than polynomially many steps or failing to decide) on all but finitely many inputs $x \in K$.

In fact, the proofs of results (A), (B), and (C) all have the same overall structure as the proof that no $\leq_m^{\mathrm{P}}$-hard language H for E is in P. In each case, a "very intractable" language $B \in \mathrm{E}$ is exhibited by diagonalization. This intractability of B, together with the fact that $B \leq_m^{\mathrm{P}} H$, is then shown to imply the appropriate intractability property for H.

At this time, it appears likely that most interesting intractable problems are not $\leq_m^{\mathrm{P}}$-hard for E or larger classes. Insofar as this is true, results such as (A), (B), and (C) fail to have interesting cases. Lutz [8] proposed to remedy this limitation by weakening the requirement that H be $\leq_m^{\mathrm{P}}$-hard for E in such results.

To be more specific, given a language H, the $\leq_m^{\mathrm{P}}$-*span* of H (also called the *lower* $\leq_m^{\mathrm{P}}$-*span* of H [7]) is the set

$$\mathrm{P}_m(H) = \left\{ A \subseteq \{0,1\}^* \mid A \leq_m^{\mathrm{P}} H \right\}$$

consisting of all languages that are polynomial time many-one reducible to H. The language H is $\leq_m^{\mathrm{P}}$-hard for E if $\mathrm{E} \subseteq \mathrm{P}_m(H)$, i.e., if $\mathrm{P}_m(H)$ contains *all* of the complexity class E. Lutz [8] proposed consideration of weaker hypotheses, stating only that $\mathrm{P}_m(H)$ contains a nonnegligible subset of E.

The expression "nonnegligible subset of E" can be assigned two useful meanings, one in terms of *resource-bounded category* [8] and the other in terms of *resource-bounded measure* [9], [10]. (*Caution:* Resource-bounded measure was incorrectly formulated in [8]. The present paper refers only to the corrected formulation in terms of martingales presented in [9], [10] and discussed briefly in §3.) Resource-bounded category, a complexity-theoretic generalization of classical Baire category [17], led to an extension of result (B) in [8]. Work since [8] has focused instead on resource-bounded measure.

Resource-bounded measure is a generalization of classical Lebesgue measure [2], [18], [17]. As such, it has Lebesgue measure as a special case, but other special cases provide internal measures for various complexity classes. This paper concerns the special case of measure in the complexity class E. In particular, resource-bounded measure defines precisely what it means for a set X of languages to have *measure* 0 *in* E. This condition, written $\mu(X \mid \mathrm{E}) = 0$, means intuitively that $X \cap \mathrm{E}$ is a *negligibly small* subset of E. (This intuition is justified technically in [9] and in §3.) A set Y of languages has *measure* 1 *in* E, written $\mu(Y \mid \mathrm{E}) = 1$, if $\mu(Y^c \mid \mathrm{E}) = 0$, where Y^c is the complement of Y. In this latter case, Y is said to contain *almost every* language in E.

It is emphasized here that not every set X of languages has a measure ("is measurable") in E. In particular, the expression "$\mu(X \mid \mathrm{E}) \neq 0$" only means that X does not have measure 0 in E. It does not necessarily imply that X has some other measure in E.

Generalizing the notion of $\leq_m^{\mathrm{P}}$-hardness for E, say that a language H is *weakly* $\leq_m^{\mathrm{P}}$-*hard* for E if $\mu(\mathrm{P}_m(H) \mid \mathrm{E}) \neq 0$, i.e., if $\mathrm{P}_m(H)$ does not have measure 0 in E. Similarly, say that a language C is *weakly* $\leq_m^{\mathrm{P}}$-*complete* for E if $C \in \mathrm{E}$ and C is weakly $\leq_m^{\mathrm{P}}$-hard for E. Since E does not have measure 0 in E [9], it is clear that every $\leq_m^{\mathrm{P}}$-hard language for E is weakly $\leq_m^{\mathrm{P}}$-hard for E, and hence every $\leq_m^{\mathrm{P}}$-complete language for E is weakly $\leq_m^{\mathrm{P}}$-complete for E.

The following extensions of results (A), (B), and (C) are now known:

(A′) Lutz and Mayordomo [12] have shown that every weakly $\leq_m^{\mathrm{P}}$-hard language H for E (in fact, every $\leq_{n^\alpha\text{-}tt}^{\mathrm{P}}$-hard language for E for $\alpha < 1$) is dense.

(B′) The method of [12] extends in a straightforward manner to show that, for every weakly $\leq_m^{\mathrm{P}}$-hard language H for E and every language $A \in \mathrm{P}$, the symmetric difference $A \triangle H$ is dense.

(C′) Juedes and Lutz [7] have shown that every weakly $\leq_m^{\mathrm{P}}$-hard language H for E has a dense exponential complexity core K. (This condition, defined in §2, implies immediately that K is a dense polynomial complexity core of H.)

Results (A′), (B′), and (C′) extend the strong intractability results (A), (B), and (C) from $\leq_m^{\mathrm{P}}$-hard languages for E to weakly $\leq_m^{\mathrm{P}}$-hard languages for E. This extends the class of problems to which well-understood lower bound techniques can be applied *unless every weakly* $\leq_m^{\mathrm{P}}$*-hard language for* E *is already* $\leq_m^{\mathrm{P}}$*-hard for* E. Surprisingly, although weak $\leq_m^{\mathrm{P}}$-hardness appears to be a weaker hypothesis than $\leq_m^{\mathrm{P}}$-hardness, this has not been proven to date.

The present paper remedies this situation. In fact, the main theorem, in §4, says that there exist languages that are weakly $\leq_m^{\mathrm{P}}$-complete but not $\leq_m^{\mathrm{P}}$-complete for E. It follows that results (A′), (B′), and (C′) do indeed extend the class of problems for which strong intractability results can be proven.

The main theorem is proven by means of a new diagonalization method called *martingale diagonalization*. This method involves the simultaneous construction, by a mutual recursion, of (i) an infinite sequence of polynomial time computable martingales (betting strategies), and (ii) a corresponding sequence of languages that defeats these martingales (prevents them from winning too much money), while also pursuing another agenda. The interplay between these two constructions ensures that the sequence of languages in (ii) can be used to construct a language that is weakly $\leq_m^{\mathrm{P}}$-complete but not $\leq_m^{\mathrm{P}}$-complete for E. Martingale diagonalization may turn out to be useful for a variety of applications.

The proof of the main theorem also makes essential use of a recent theorem of Juedes and Lutz [7], which gives a nontrivial *upper* bound on the complexities of $\leq_m^{\mathrm{P}}$-hard languages for E.

Section 2 presents basic notation and definitions. Section 3 provides definitions and basic properties of feasible (polynomial time computable) martingales, uses these to define measure in E, and proves a new result, the rigid enumeration theorem. This result provides a uniform enumeration of feasible martingales that is crucial for the martingale diagonalization method. Section 4 is devoted entirely to the main theorem and its proof. Section 5 briefly discusses directions for future work with particular emphasis on the search for *natural* problems that are weakly $\leq_m^{\mathrm{P}}$-hard for E.

2. Preliminaries. All *languages* (synonymously, *decision problems*) in this paper are sets of binary strings, i.e., sets $A \subseteq \{0, 1\}^*$.

The *standard enumeration* of $\{0, 1\}^*$ is the infinite sequence

$$\lambda, 0, 1, 00, 01, 10, 11, 000, 001, \ldots$$

in which strings appear first in order of length, then in lexicographic order. The symbol λ denotes the *empty string* and the expression $|w|$ denotes the *length* of a string $w \in \{0, 1\}^*$. It is convenient to write the standard enumeration in the form

$$\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \ldots.$$

That is, for each $n \in \mathbf{N}$, $\mathbf{n}$ is the nth string (counting from 0) in the standard enumeration of $\{0, 1\}^*$. Thus, $\mathbf{0} = \lambda$, $\mathbf{1} = 0$, $\mathbf{2} = 1$, $\mathbf{3} = 00$, etc. Note also that $|\mathbf{n}|$ denotes the length of the nth string in $\{0, 1\}^*$.

The *Boolean value* of a condition ψ is

$$[\![\psi]\!] = \begin{cases} 1 & \text{if } \psi \text{ is true,} \\ 0 & \text{if } \psi \text{ is false.} \end{cases}$$

Each language $A \subseteq \{0, 1\}^*$ is identified with its *characteristic sequence*, which is the infinite binary sequence

$$\chi_A = [\![\mathbf{0} \in A]\!][\![\mathbf{1} \in A]\!][\![\mathbf{2} \in A]\!] \ldots .$$

The expression $\chi_A[0..n-1]$ denotes the string consisting of the first n bits of χ_A.

This paper uses the standard *pairing function*

$$\langle , \rangle : \mathbf{N} \times \mathbf{N} \xrightarrow[\text{onto}]{1\text{-}1} \mathbf{N}$$

defined by

$$\langle k, n \rangle = \binom{k+n+1}{2} + k$$

for all $k, n \in \mathbf{N}$. This pairing function induces the pairing function

$$\langle , \rangle : \{0, 1\}^* \times \{0, 1\}^* \xrightarrow[\text{onto}]{1\text{-}1} \{0, 1\}^*$$

defined in the obvious way, i.e., $\langle \mathbf{k}, \mathbf{n} \rangle$ is the $\langle k, n \rangle$th string in the standard enumeration of $\{0, 1\}^*$. Note that $|\langle \mathbf{k}, \mathbf{n} \rangle| \leq 2(|\mathbf{k}| + |\mathbf{n}|)$ for all $\mathbf{k}, \mathbf{n} \in \{0, 1\}^*$.

As noted in §1, a language $A \subseteq \{0, 1\}^*$ is *dense* if there is a real number $\varepsilon > 0$ such that, for all sufficiently large n, A contains at least 2^{n^ε} strings $x \in \{0, 1\}^{\leq n}$.

Given a function $t : \mathbf{N} \to \mathbf{N}$, the complexity class DTIME($t(n)$) consists of every language $A \subseteq \{0, 1\}^*$ such that $[\![x \in A]\!]$ is computable (by a deterministic Turing machine) in $O(t(|x|))$ steps. Similarly, the complexity class DTIMEF($t(n)$) consists of every function $f : \{0, 1\}^* \to \{0, 1\}^*$ such that $f(x)$ is computable in $O(t(|x|))$ steps. The complexity classes

$$\mathrm{P} = \bigcup_{k=0}^{\infty} \mathrm{DTIME}(n^k),$$

$$\mathrm{PF} = \bigcup_{k=0}^{\infty} \mathrm{DTIMEF}(n^k),$$

$$\mathrm{E} = \bigcup_{k=0}^{\infty} \mathrm{DTIME}(2^{kn}),$$

$$\mathrm{E}_2 = \bigcup_{k=0}^{\infty} \mathrm{DTIME}(2^{n^k})$$

are of particular interest in this paper.

A language A is *polynomial time many-one reducible* to a language B, written $A \leq_m^{\mathrm{P}} B$, if there is a function $f \in \mathrm{PF}$ such that $A = f^{-1}(B)$, i.e., for all $x \in \{0, 1\}^*$, $x \in A \iff f(x) \in B$.

Complexity cores, first introduced by Lynch [13], have been studied extensively. The rest of this section specifies the notions of complexity cores mentioned in §1.

Given a (deterministic Turing) machine M and an input $x \in \{0, 1\}^*$, write

$$M(x) = \begin{cases} 1 & \text{if } M \text{ accepts } x, \\ 0 & \text{if } M \text{ rejects } x, \\ \perp & \text{in any other case.} \end{cases}$$

If $M(x) \in \{0, 1\}$, then $\text{time}_M(x)$ denotes the number of steps used in the computation of $M(x)$. If $M(x) = \perp$, then $\text{time}_M(x) = \infty$. A machine M is *consistent* with a language A if $M(x) = [\![x \in A]\!]$ whenever $M(x) \in \{0, 1\}$.

DEFINITION. *Let $t : \mathbf{N} \to \mathbf{N}$ be a time bound and let $A, K \subseteq \{0, 1\}^*$. Then K is a* DTIME($t(n)$)-complexity core *of A if, for every $c \in \mathbf{N}$ and every machine M that is consistent with A, the "fast set"*

$$F = \{ x \mid \text{time}_M(x) \leq c \cdot t(|x|) + c \}$$

has finite intersection with K. (By the definition of $\text{time}_M(x)$, $M(x) \in \{0, 1\}$ for all $x \in F$. Thus F is the set of all strings that M "decides efficiently.")

Note that every subset of a DTIME($t(n)$)-complexity core of A is a DTIME($t(n)$)-complexity core of A. Note also that, if $s(n) = O(t(n))$, then every DTIME($t(n)$)-complexity core of A is a DTIME($s(n)$)-complexity core of A.

DEFINITION. *Let $A, K \subseteq \{0, 1\}^*$.*

1. *K is a* polynomial complexity core *of A if K is a* DTIME(n^k)*-complexity core of A for all $k \in \mathbf{N}$.*

2. *K is an* exponential complexity core *of A if there is a real number $\epsilon > 0$ such that K is a* DTIME(2^{n^ϵ})*-complexity core of A.*

Intuitively, a P-complexity core of A is a set of infeasible instances of A, while an exponential complexity core of A is a set of extremely hard instances of A.

3. Feasible martingales. This section presents some basic properties of martingales (betting strategies) that are computable in polynomial time. Such martingales are used to develop a fragment of resource-bounded measure that is sufficient for understanding the notion of weakly hard problems. This section also proves the rigid enumeration theorem, which is crucial for the martingale diagonalization method used to prove the main theorem in §4.

DEFINITION. *A* martingale *is a function $d : \{0, 1\}^* \to [0, \infty)$ with the property that, for all $w \in \{0, 1\}^*$,*

$$d(w) = \frac{d(w0) + d(w1)}{2}. \tag{3.1}$$

A martingale d succeeds *on a language $A \subseteq \{0, 1\}^*$ if*

$$\limsup_{n \to \infty} d(\chi_A[0..n-1]) = \infty.$$

(Recall that $\chi_A[0..n-1]$ is the string consisting of the first n bits of the characteristic sequence of A.) Finally, for each martingale d, define the set

$$S^\infty[d] = \left\{ A \subseteq \{0, 1\}^* \mid d \text{ succeeds on } A \right\}.$$

Intuitively, a martingale d is a betting strategy that, given a language A, starts with capital (amount of money) $d(\lambda)$ and bets on the membership or nonmembership of the successive strings $\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots$ (the standard enumeration of $\{0, 1\}^*$) in A. Prior to betting on a string $\mathbf{n}$, the strategy has capital $d(w)$, where

$$w = [\![\mathbf{0} \in A]\!] \ldots [\![\mathbf{n - 1} \in A]\!].$$

After betting on the string $\mathbf{n}$, the strategy has capital $d(wb)$, where $b = [\![\mathbf{n} \in A]\!]$. Condition (3.1) ensures that the betting is fair. The strategy succeeds on A if its capital is unbounded as the betting progresses.

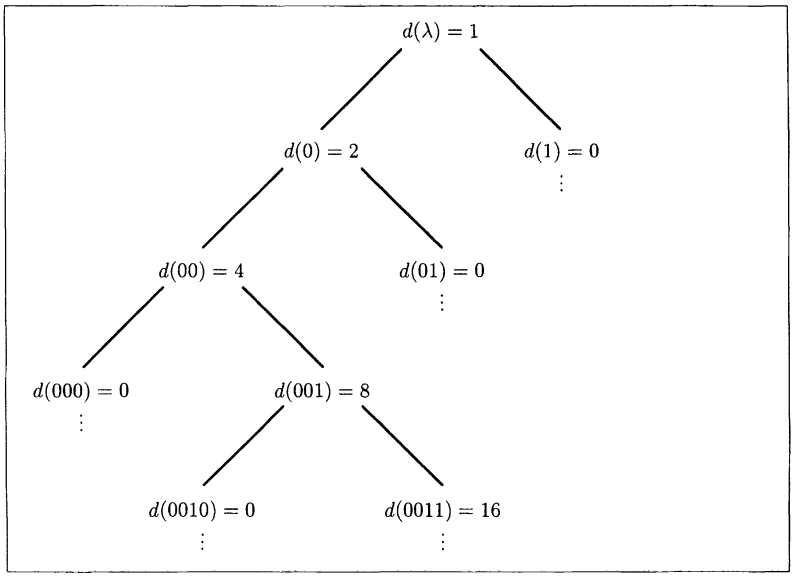


FIG. 1. *The martingale d of Example* 3.1.

Example 3.1. Define $d : \{0, 1\}^* \to [0, \infty)$ by the following recursion. Let $w \in \{0, 1\}^*$ and $b \in \{0, 1\}$.

(i) $d(\lambda) = 1$.

(ii) $d(wb) = 2 \cdot d(w) \cdot [\![b = [\![\, |w| \text{ is prime}]\!]]\!]$.

(See Fig. 1.) It is easily checked that d is a martingale that succeeds on the language $A = \{\, \mathbf{p} \mid p \text{ is prime}\,\}$ and on no other language.

Example 3.2. Define $d : \{0, 1\}^* \to [0, \infty)$ by the following recursion. Let $w \in \{0, 1\}^*$.

(i) $d(\lambda) = 1$.

(ii) $d(w0) = \frac{3}{2}d(w)$.

(iii) $d(w1) = \frac{1}{2}d(w)$.

(See Fig. 2.) It is obvious that d is a martingale that succeeds on every finite language A. In fact, it is easily checked that $S^\infty[d]$ contains exactly every language A for which the quantity

$$\#(0, \chi_A[0..n-1]) - \frac{n}{\log 3}$$

is unbounded as $n \to \infty$, where $\#(0, w)$ denotes the number of 0's in the string w.

Martingales were used extensively by Schnorr [19]–[22] in his investigation of random and pseudorandom sequences. Lutz [9], [10] used martingales that are computable in polynomial time to characterize sets that have measure 0 in E.

Since martingales are real valued, their computations must employ finite approximations of real numbers. For this purpose, let

$$\mathbf{D} = \left\{\, m \cdot 2^{-n} \;\middle|\; m, n \in \mathbf{N} \right\}$$

be the set of *nonnegative dyadic rationals.* These are nonnegative rational numbers with finite binary expansions.

DEFINITION. 1. *A* computation *of a martingale d is a function* $\widehat{d} : \mathbf{N} \times \{0, 1\}^* \to \mathbf{D}$ *such that*

$$\left|\widehat{d}_r(w) - d(w)\right| \leq 2^{-r} \tag{3.2}$$

for all $r \in \mathbf{N}$ *and* $w \in \{0, 1\}^*$ *satisfying* $r \geq |w|$, *where* $\widehat{d}_r(w) = \widehat{d}(r, w)$.

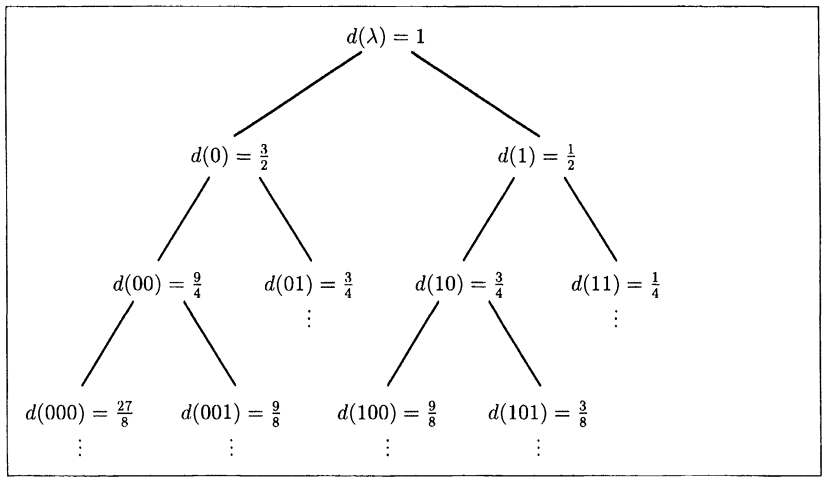


FIG. 2. *The martingale d of Example* 3.2.

2. *A* strong computation *of a martingale d is a computation* $\widehat{d}$ *of d that satisfies* (3.2) *for* all $r \in \mathbf{N}$ *and* $w \in \{0, 1\}^*$.

3. *A computation* $\widehat{d}$ *of a martingale d is* rigid *if it has the following two properties:*

(a) *For each* $r \in \mathbf{N}$, *the function* $\widehat{d}_r$ *is a martingale.*

(b) *For all* $r \in \mathbf{N}$ *and* $w \in \{0, 1\}^*$, if $r \geq |w|$, *then*

$$\left|\widehat{d}_r(w) - \widehat{d}_{r+1}(w)\right| \leq 2^{-(r+1)}.$$

4. *A* p-computation *of a martingale d is a computation* $\widehat{d}$ *of d such that* $\widehat{d}_r(w)$ *is computable in time polynomial in* $r + |w|$.

5. *A* p-martingale *is a martingale that has a* p-*computation.*

Here, a martingale is considered "feasible" if and only if it is a p-martingale, i.e., if and only if it has a p-computation. Intuitively, one might prefer to insist that feasible martingales have *strong* p-computations, thereby avoiding the ad hoc condition $r \geq |w|$. On the other hand, in the technical arguments of this paper, it is useful to have *rigid* p-computations for reasons explained below. Fortunately, the following lemma shows that all three of these conditions are equivalent.

LEMMA 3.3 (rigid computation lemma). *For a martingale d, the following three conditions are equivalent:*

(1) *d has a* p-*computation.*

(2) *d has a strong* p-*computation.*

(3) *d has a rigid* p-*computation.*

Proof. It is trivial that (3) implies (1). To see that (1) implies (2), let $\widehat{d}$ be a p-computation of d. Then the function $\widetilde{d} : \mathbf{N} \times \{0, 1\}^* \to \mathbf{D}$ defined by $\widetilde{d}_r(w) = \widehat{d}_{r+|w|}(w)$ is easily seen to be a strong p-computation of d, so (2) holds.

To see that (2) implies (3), let $\widehat{d}$ be a strong p-computation of d. Define a function $\widetilde{d} : \mathbf{N} \times \{0, 1\}^* \to \mathbf{D}$ by the following recursion. Assume that $r \in \mathbf{N}$, $w \in \{0, 1\}^*$, $b \in \{0, 1\}$, and $\overline{b} = 1 - b$.

(i) $\widetilde{d}_r(\lambda) = \widehat{d}_{2r+2}(\lambda)$.

(ii) $\widetilde{d}_r(wb) = \widetilde{d}_r(w) + (\widehat{d}_{2r+2}(wb) - \widehat{d}_{2r+2}(w\overline{b}))/2$.

It suffices to show that $\widetilde{d}$ is a rigid p-computation of d.

It is first shown, by induction on w, that

$$|\widetilde{d}_r(w) - d(w)| \leq 2^{-(2r+2)}(1 + |w|) \tag{3.3}$$

holds for all $r \in \mathbf{N}$ and $w \in \{0, 1\}^*$. For $w = \lambda$, this follows immediately from the facts that $\widetilde{d}_r(\lambda) = \widehat{d}_{2r+2}(\lambda)$ and $\widehat{d}$ is a p-computation of d. For the induction step, assume that (3.3) holds. Then, for $b \in \{0, 1\}$,

$$\begin{aligned}
|\widetilde{d}_r(wb) - d(wb)| &= \left|\widetilde{d}_r(w) + \frac{\widehat{d}_{2r+2}(wb) - \widehat{d}_{2r+2}(w\bar{b})}{2} - d(wb)\right| \\
&\leq |\widetilde{d}_r(w) - d(w)| + \left|d(w) + \frac{\widehat{d}_{2r+2}(wb) - \widehat{d}_{2r+2}(w\bar{b})}{2} - d(wb)\right| \\
&= |\widetilde{d}_r(w) - d(w)| + \left|\frac{d(wb) + d(w\bar{b})}{2} + \frac{\widehat{d}_{2r+2}(wb) - \widehat{d}_{2r+2}(w\bar{b})}{2} - d(wb)\right| \\
&= |\widetilde{d}_r(w) - d(w)| + \left|\frac{\widehat{d}_{2r+2}(wb) - d(wb)}{2} + \frac{d(w\bar{b}) - \widehat{d}_{2r+2}(w\bar{b})}{2}\right| \\
&\leq |\widetilde{d}_r(w) - d(w)| + \frac{1}{2}|\widehat{d}_{2r+2}(wb) - d(wb)| + \frac{1}{2}|\widehat{d}_{2r+2}(w\bar{b}) - d(w\bar{b})| \\
&\leq 2^{-(2r+2)}(1 + |w|) + 2^{-(2r+2)} \\
&= 2^{-(2r+2)}(1 + |wb|).
\end{aligned}$$

(The last inequality holds by the induction hypothesis and the fact that $\widehat{d}$ is a strong p-computation of d.) This confirms the fact that (3.3) holds for all $r \in \mathbf{N}$ and $w \in \{0, 1\}^*$.

Now let $r \in \mathbf{N}$ and $w \in \{0, 1\}^*$ be such that $r \geq |w|$. Then, by (3.3),

$$\begin{aligned}
|\widetilde{d}_r(w) - d(w)| &\leq 2^{-(2r+2)}(1 + |w|) \\
&\leq 2^{-(2r+2)}(1 + r) \\
&\leq 2^{-(r+2)}.
\end{aligned} \tag{3.4}$$

This shows that $\widetilde{d}$ is a computation of d. In fact, since $\widehat{d}$ is a p-computation, the fact that $\widetilde{d}$ is a p-computation of d is easily checked. The fact that $\widetilde{d}$ is rigid follows from the following two observations:

(a) For each $r \in \mathbf{N}$, the function $\widetilde{d}_r$ is clearly a martingale by clause (ii) in the definition of $\widetilde{d}$.

(b) For all $r \in \mathbf{N}$ and $w \in \{0, 1\}^*$, by (3.4),

$$\begin{aligned}
|\widetilde{d}_r(w) - \widetilde{d}_{r+1}(w)| &\leq |\widetilde{d}_r(w) - d(w)| + |\widetilde{d}_{r+1}(w) - d(w)| \\
&\leq 2^{-(r+2)} + 2^{-(r+3)} \\
&< 2^{-(r+1)}.
\end{aligned}$$

Thus (3) holds. □

Note that the proof of Lemma 3.3 does not construct a p-computation of d that is both strong and rigid. In fact, it seems reasonable to conjecture that there exists a p-martingale d for which no p-computation is both strong and rigid.

Note that a function $\widehat{d} : \mathbf{N} \times \{0,1\}^* \to \mathbf{D}$ is a rigid computation of *some* martingale d if and only if it satisfies the predicates

$$\alpha_{r,w}(\widehat{d}) \equiv \left[r < |w| \text{ or } \left|\widehat{d}_r(w) - \widehat{d}_{r+1}(w)\right| \leq 2^{-(r+1)}\right]$$

and

$$\beta_{r,w}(\widehat{d}) \equiv \left[\widehat{d}_r(w) = \frac{\widehat{d}_r(w0) + \widehat{d}_r(w1)}{2}\right]$$

for all $r \in \mathbf{N}$ and $w \in \{0,1\}^*$. The next theorem exploits this fact to give a very useful enumeration of all p-martingales. The following definition specifies the useful properties of this enumeration.

DEFINITION. *A* rigid enumeration $d_0, d_1, \ldots; \widehat{d}_0, \widehat{d}_1, \ldots$ *of all* p*-martingales consists of a sequence* $d_0, d_1, \ldots$ *and a sequence* $\widehat{d}_0, \widehat{d}_1, \ldots$ *with the following properties:*

(i) $d_0, d_1, \ldots$ *is an enumeration of all* p*-martingales.*

(ii) *For each* $k \in \mathbf{N}$, $\widehat{d}_k$ *is a rigid* p*-computation of* d_k.

(iii) *There is an algorithm that, given* $k, r \in \mathbf{N}$ *and* $w \in \{0,1\}^*$, *computes* $\widehat{d}_{k,r}(w)$ *in at most* $(2 + r + |w|)^{|\mathbf{k}|}$ *steps.*

The following theorem is the main result of this section.

THEOREM 3.4 (rigid enumeration theorem). *There exists a rigid enumeration of all p-martingales.*

Proof. Fix a function $\widetilde{g} : \mathbf{N}^2 \times \{0,1\}^* \to \mathbf{D}$ with the following properties: (Write $\widetilde{g}_{k,r}(w) = \widetilde{g}_k(r,w) = \widetilde{g}(k,r,w)$.)

(i) $\widetilde{g}_0, \widetilde{g}_1, \ldots$ is an enumeration of all functions $f : \mathbf{N} \times \{0,1\}^* \to \mathbf{D}$ such that $f(r,w)$ is computable in time polynomial in $r + |w|$.

(ii) There is an algorithm that, given $k, r \in \mathbf{N}$ and $w \in \{0,1\}^*$, computes $\widetilde{g}_{k,r}(w)$ in at most $(2 + r + |w|)^{|\mathbf{k}|}$ steps.

(The existence of such an efficient universal function is well known [3], [4].)

Most of this proof is devoted to two claims and their respective proofs.

CLAIM 1. *There is a function* $\widehat{g} : \mathbf{N}^2 \times \{0,1\}^* \to \mathbf{D}$ *with the following properties: (Write* $\widehat{g}_{k,r}(w) = \widehat{g}_k(r,w) = \widehat{g}(k,r,w)$.)

(a) *For each* $k \in \mathbf{N}$, $\widehat{g}_k$ *is a rigid* p*-computation of some martingale* g_k.

(b) *For each* $k \in \mathbf{N}$, *if* $\widetilde{g}_k$ *is already a rigid* p*-computation of some martingale* g_k, *then* $\widehat{g}_k = \widetilde{g}_k$.

(c) *There is a constant* $c \in \mathbf{N}$ *such that, for all* $k, r \in \mathbf{N}$ *and* $w \in \{0,1\}^*$, $\widehat{g}_{k,r}(w)$ *is computable in at most* $(2 + r + |w|)^{c+c\cdot|\mathbf{k}|}$ *steps.*

Assume for the moment that Claim 1 is true. Define functions $\widehat{d} : \mathbf{N}^2 \times \{0,1\}^* \to \mathbf{D}$ and $d : \mathbf{N} \times \{0,1\}^* \to [0,\infty)$ by

$$\widehat{d}_{k,r}(w) = \begin{cases} \widehat{g}_{j,r}(w) & \text{if } \mathbf{k} = 0^{c\cdot(1+|\mathbf{j}|)}1\mathbf{j}, \\ 0 & \text{if } \mathbf{k} \text{ is not of this form,} \end{cases}$$

$$d_k(w) = \lim_{r\to\infty} \widehat{d}_{k,r}(w).$$

CLAIM 2. *The sequences* $d_0, d_1, \ldots$ *and* $\widehat{d}_0, \widehat{d}_1, \ldots$ *constitute a rigid enumeration of all* p*-martingales.*

To prove Claim 2 (still assuming Claim 1), first note that, for all $k \in \mathbf{N}$ and $w \in \{0, 1\}^*$,

$$d_k(w) = \begin{cases} g_j(w) & \text{if } \mathbf{k} = 0^{c\cdot(1+|\mathbf{j}|)}1\mathbf{j}, \\ 0 & \text{if } \mathbf{k} \text{ is not of this form.} \end{cases}$$

By part (a) of Claim 1, this immediately implies that each d_k is a p-martingale. Conversely, assume that $d' : \{0, 1\}^* \to [0, \infty)$ is a p-martingale. Then, by the rigid computation lemma and clause (i) in the specification of $\widetilde{g}$, there is some $j \in \mathbf{N}$ such that $\widetilde{g}_j$ is a rigid p-computation of d'. Choose $k \in \mathbf{N}$ such that $\mathbf{k} = 0^{c\cdot(1+|\mathbf{j}|)}1\mathbf{j}$. Then $\widehat{d}_k = \widehat{g}_j = \widetilde{g}_j$ by part (b) of Claim 1, so $\widehat{d}_k$ is a rigid p-computation of d', so $d_k = d'$. This shows that $d_0, d_1, \ldots$ is an enumeration of all p-martingales and each $\widehat{d}_k$ is a rigid p-computation of d_k. For $\mathbf{k} = 0^{c\cdot(1+|\mathbf{j}|)}1\mathbf{j}$, the time $t(k, r, w)$ required to compute $\widehat{d}_{k,r}(w)$ satisfies

$$\begin{aligned} t(k, r, w) &\leq |\mathbf{k}| + (2 + r + |w|)^{c\cdot(1+|\mathbf{j}|)} \\ &\leq 2^{|\mathbf{k}|-1} + (2 + r + |w|)^{|\mathbf{k}|-1} \\ &\leq (2 + r + |w|)^{|\mathbf{k}|}. \end{aligned}$$

This proves Claim 2, and hence the theorem. All that remains, then, is to prove Claim 1.

To prove Claim 1, the values $\widehat{g}_{k,r}(w)$ are first specified for all $k, r \in \mathbf{N}$ and $w \in \{0, 1\}^*$. Define the following predicates: (In these predicates, it is useful to regard $k, r \in \mathbf{N}$ and $w \in \{0, 1\}^*$ as parameters and $f, \widehat{f} : \mathbf{N}^2 \times \{0, 1\}^* \to \mathbf{D}$ as variables.)

$$\alpha_{k,r,w}(f, \widehat{f}) \equiv \left[r < |w| \text{ or } \left|\widehat{f}_{k,r}(w) - f_{k,r+1}(w)\right| \leq 2^{-(r+1)}\right],$$

$$\beta_{k,r,w}(f, \widehat{f}) \equiv \left[\widehat{f}_{k,r}(w) = \frac{f_{k,r}(w0) + f_{k,r}(w1)}{2}\right].$$

Define $\widehat{g} : \mathbf{N}^2 \times \{0, 1\}^* \to \mathbf{D}$ by recursion on r and w as follows. Let $k, r \in \mathbf{N}$, $w \in \{0, 1\}^*$, and $b \in \{0, 1\}$.

$$\text{(I)} \quad \widehat{g}_{k,0}(\lambda) = \widetilde{g}_{k,0}(\lambda).$$

$$\text{(II)} \quad \widehat{g}_{k,r+1}(\lambda) = \begin{cases} \widetilde{g}_{k,r+1}(\lambda) & \text{if } \alpha_{k,r,\lambda}(\widetilde{g}, \widehat{g}), \\ \widehat{g}_{k,r}(\lambda) & \text{otherwise.} \end{cases}$$

$$\text{(III)} \quad \widehat{g}_{k,0}(wb) = \begin{cases} \widetilde{g}_{k,0}(wb) & \text{if } \beta_{k,0,w}(\widetilde{g}, \widehat{g}), \\ \widehat{g}_{k,0}(w) & \text{otherwise.} \end{cases}$$

$$\text{(IV)} \quad \widehat{g}_{k,r+1}(wb) = \begin{cases} \widetilde{g}_{k,r+1}(wb) & \text{if } \alpha_{k,r,w0}(\widetilde{g}, \widehat{g}) \text{ and} \\ & \quad \alpha_{k,r,w1}(\widetilde{g}, \widehat{g}) \text{ and} \\ & \quad \beta_{k,r+1,w}(\widetilde{g}, \widehat{g}), \\ \widehat{g}_{k,r}(wb) + \widehat{g}_{k,r+1}(w) - \widehat{g}_{k,r}(w) & \text{otherwise.} \end{cases}$$

By condition (ii) in the choice of $\widetilde{g}$, the function $\widehat{g}$ defined by this recursion is easily seen to satisfy condition (c) of Claim 1.

To see that $\widehat{g}$ satisfies condition (a) of Claim 1, let $k \in \mathbf{N}$ be arbitrary. A routine induction on r shows that $\beta_{k,r,w}(\widehat{g}, \widehat{g})$ holds for all $r \in \mathbf{N}$ and $w \in \{0,1\}^*$. It follows easily that each $\widehat{g}_{k,r}$ is a martingale. A routine induction on w then shows that $\alpha_{k,r,w}(\widehat{g}, \widehat{g})$ holds for all $r \in \mathbf{N}$ and $w \in \{0,1\}^*$. It follows that $\widehat{g}_k$ is a rigid p-computation of the martingale g_k defined by $g_k(w) = \lim_{r\to\infty} \widehat{g}_{k,r}(w)$. Thus $\widehat{g}$ satisfies condition (a) of Claim 1.

Finally, to see that $\widehat{g}$ satisfies condition (b) of Claim 1, fix $k \in \mathbf{N}$ and assume that $\widetilde{g}_k$ is a rigid computation of some martingale g_k. Then a routine induction on r and w shows that $\widehat{g}_k = \widetilde{g}_k$. (The α and β predicates hold throughout the induction, so the "otherwise" cases are never invoked in the definition of $\widehat{g}_k$.) This completes the proof of Claim 1 and the proof of the rigid enumeration theorem. □

The rest of this section briefly develops those aspects of measure in E that are used in this paper. The key ideas are in the following definition.

DEFINITION. 1. *A set X of languages has* p-measure 0, *written* $\mu_{\mathrm{p}}(X) = 0$, *if there is a* p-*martingale d such that $X \subseteq S^\infty[d]$.*

2. *A set X of languages has* measure 0 in E, *written* $\mu(X \mid \mathrm{E}) = 0$, *if* $\mu_{\mathrm{p}}(X \cap \mathrm{E}) = 0$.

3. *A set X of languages has* measure 1 in E, *written* $\mu(X \mid \mathrm{E}) = 1$, if $\mu(X^c \mid \mathrm{E}) = 0$, *where X^c is the complement of X. In this case, X is said to contain* almost every *language in* E.

4. *The expression* $\mu(X \mid \mathrm{E}) \neq 0$ *indicates that X does* not *have measure* 0 *in* E. *Note that this does* not *assert that* $\mu(X \mid \mathrm{E})$ *has some nonzero value.*

Thus, a set X of languages has measure 0 in E if there is a feasible martingale that succeeds on every element of X.

The following fact is obvious but useful.

PROPOSITION 3.5. *Every set X of languages satisfies the implications*

$$\mu_{\mathrm{p}}(X) = 0 \Longrightarrow \mu(X \mid \mathrm{E}) = 0, \qquad \mu_{\mathrm{p}}(X) = 0 \Longrightarrow \Pr[A \in X] = 0,$$

where the probability $\Pr[A \in X]$ *is computed according to the random experiment in which a language $A \subseteq \{0,1\}^*$ is chosen probabilistically using an independent toss of a fair coin to decide whether each string $x \in \{0,1\}^*$ is in A.*

The right-hand implication in Proposition 3.5 makes it clear that p-measure 0 sets are negligibly small. What is significant for complexity theory is that, if X has measure 0 in E, then $X \cap \mathrm{E}$ is negligibly small *as a subset of* E. This intuition is technically justified in [9], where it is shown that finite subsets of E have measure 0 in E and the sets of measure 0 in E are closed under subset, finite unions, and certain countable unions called p-unions. Most importantly, the following theorem is shown.

THEOREM 3.6 [9]. $\mu(\mathrm{E} \mid \mathrm{E}) \neq 0$.

Combined with the aforementioned closure properties, this result (which is a special case of the more general measure conservation theorem [9]) ensures that $X \cap \mathrm{E}$ is, in a nontrivial sense, a negligibly small subset of E whenever X has measure 0 in E.

4. Weak completeness in E. In standard terminology, a language H is $\leq_m^{\mathrm{P}}$-*hard* for a complexity class $\mathcal{C}$ if the set

$$\mathrm{P}_m(H) = \left\{ A \;\middle|\; A \leq_m^{\mathrm{P}} H \right\}$$

contains all of $\mathcal{C}$. A language C is $\leq_m^{\mathrm{P}}$-*complete* for $\mathcal{C}$ if $C \in \mathcal{C}$ and C is $\leq_m^{\mathrm{P}}$-hard for $\mathcal{C}$. The following definition generalizes these notions for the complexity class $\mathcal{C} = \mathrm{E}$.

DEFINITION. *A language H is* weakly $\leq_m^{\mathrm{P}}$-hard *for* E *if* $\mu(\mathrm{P}_m(H) \mid \mathrm{E}) \neq 0$, i.e., *the set* $\mathrm{P}_m(H)$ *does not have measure* 0 *in* E. *A language C is* weakly $\leq_m^{\mathrm{P}}$-complete *for* E *if* $C \in \mathrm{E}$ *and C is weakly* $\leq_m^{\mathrm{P}}$*-hard for* E.

By Theorem 3.6, every $\leq_m^{\mathrm{P}}$-hard language for E is weakly $\leq_m^{\mathrm{P}}$-hard for E, whence every $\leq_m^{\mathrm{P}}$-complete language for E is weakly $\leq_m^{\mathrm{P}}$-complete for E. The following result says that the converse does *not* hold, i.e., that in E, weak $\leq_m^{\mathrm{P}}$-completeness is a proper generalization of $\leq_m^{\mathrm{P}}$-completeness.

THEOREM 4.1 (main theorem). *There is a language C that is weakly* $\leq_m^{\mathrm{P}}$*-complete, but not* $\leq_m^{\mathrm{P}}$*-complete, for E.*

The rest of this section is devoted to proving the main theorem.

A recent theorem of Juedes and Lutz gives a necessary condition for a language to be $\leq_m^{\mathrm{P}}$-hard for E. This condition, based on an idea of Meyer [15], plays an important role in the present proof. The key ideas are developed in the following definitions.

DEFINITION. *The* collision set *of a function* $f : \{0,1\}^* \to \{0,1\}^*$ *is*

$$C_f = \{ n \in \mathbf{N} \mid (\exists m < n)\, f(\mathbf{m}) = f(\mathbf{n}) \}.$$

A function $f : \{0,1\}^* \to \{0,1\}^*$ *is* one-to-one almost everywhere *if* C_f *is finite.*

DEFINITION. *Let* $A \subseteq \{0,1\}^*$ *and* $t : \mathbf{N} \to \mathbf{N}$. *A* many-one reduction *of A is a computable function* $f : \{0,1\}^* \to \{0,1\}^*$ *such that* $A = f^{-1}(f(A))$, *i.e., such that, for all* $x \in \{0,1\}^*$, $f(x) \in f(A)$ *implies* $x \in A$. *A* $\leq_m^{\mathrm{DTIME}(t)}$-reduction *of A is a many-one reduction f of A such that* $f \in \mathrm{DTIMEF}(t)$.

DEFINITION. *Let* $A \subseteq \{0,1\}^*$ *and* $t : \mathbf{N} \to \mathbf{N}$. *Then A is incompressible by* $\leq_m^{\mathrm{DTIME}(t)}$-reductions *if every* $\leq_m^{\mathrm{DTIME}(t)}$*-reduction of A is one-to-one almost everywhere.*

Intuitively, if f is a $\leq_m^{\mathrm{DTIME}(t)}$-reduction of A and C_f is large, then f compresses many questions ($x \in A$?) to fewer questions ($f(x) \in f(A)$?). If A is incompressible by $\leq_m^{\mathrm{DTIME}(t)}$-reductions, then A is "very complex" in the sense that very little such compression can occur.

The following result is used here.

THEOREM 4.2 (Juedes and Lutz [7]). *No language that is* $\leq_m^{\mathrm{P}}$*-hard for E is incompressible by* $\leq_m^{\mathrm{DTIME}(2^{4n})}$*-reductions.*

Since almost every language (and almost every language in E) *is* incompressible by $\leq_m^{\mathrm{DTIME}(2^{4n})}$-reductions [7], Theorem 4.2 says that the $\leq_m^{\mathrm{P}}$-hard languages are "unusually simple" in at least this one respect.

The largest part of the proof of the main theorem is the construction of a language $H \in \mathrm{E}_2$ with the following two properties:

(I) H is weakly $\leq_m^{\mathrm{P}}$-hard for E.

(II) H is incompressible by $\leq_m^{\mathrm{DTIME}(2^{4n})}$-reductions.

By Theorem 4.2, this language H cannot be $\leq_m^{\mathrm{P}}$-hard for E. A padding argument then gives the main theorem.

The language H is constructed by diagonalization. In establishing property (I), the construction uses a fixed rigid enumeration $d_0, d_1, \ldots; \widehat{d}_0, \widehat{d}_1, \ldots$ of all p-martingales. Such a rigid enumeration exists by Theorem 3.4. In establishing property (II), the construction uses a fixed function f such that $f \in \mathrm{DTIMEF}(2^{5n})$ and f is universal for $\mathrm{DTIMEF}(2^{4n})$ in the sense that

$$\mathrm{DTIMEF}(2^{4n}) = \{ f_i \mid i \in \mathbf{N} \},$$

where $f_i(x) = f(\langle \mathbf{i}, x \rangle)$. (The existence of such an efficient universal function is well known [3], [4].)

In addition to the pairing function $\langle , \rangle$ mentioned in §2, the construction of H uses the ordering $<^*$ of $\mathbf{N}^2$ defined by

$$\begin{aligned}(j,m) <^* (k,n) \iff & [(1+|\mathbf{j}|)(1+|\mathbf{m}|) < (1+|\mathbf{k}|)(1+|\mathbf{n}|) \\ & \text{or } [(1+|\mathbf{j}|)(1+|\mathbf{m}|) = (1+|\mathbf{k}|)(1+|\mathbf{n}|) \\ & \quad \text{and } \langle j,m\rangle < \langle k,n\rangle]]\end{aligned}$$

for all $j, m, k, n \in \mathbf{N}$. It is easy to check that $(\mathbf{N}^2, <^*)$ is order isomorphic to $(\mathbf{N}, <)$. For $(k,n) \in \mathbf{N}^2$, let

$$\#^*(k,n) = \left|\left\{(j,m) \in \mathbf{N}^2 \;\middle|\; (j,m) <^* (k,n)\right\}\right|$$

be the number of $<^*$-predecessors of (k,n) in $\mathbf{N}^2$. Two important properties of $<^*$ are

$$(j,m) <^* (k,n) \Longrightarrow (1+|\mathbf{j}|)(1+|\mathbf{m}|) \leq (1+|\mathbf{k}|)(1+|\mathbf{n}|)$$

and

$$\#^*(k,n) = 2^{O((1+|\mathbf{k}|)(1+|\mathbf{n}|))}.$$

Using the ordering $<^*$, define the *modified collision set* C_i^* of a function $f_i \in \mathrm{DTIMEF}(2^{4n})$ by

$$C_i^* = \left\{(k,n) \in \mathbf{N}^2 \;\middle|\; (\exists (j,m) <^* (k,n))\, f_i(\langle \mathbf{j}, \mathbf{m}\rangle) = f_i(\langle \mathbf{k}, \mathbf{n}\rangle)\right\}.$$

Also, for $k \in \mathbf{N}$, define the kth *slice* of C_i^* as the set

$$C_{i,k}^* = \left\{ n \in \mathbf{N} \;\middle|\; (k,n) \in C_i^* \right\}.$$

LEMMA 4.3. *For all $i \in \mathbf{N}$, the function f_i is one-to-one almost everywhere if and only if the set C_i^* is finite.*

Proof. Fix $i \in \mathbf{N}$ and define an equivalence relation $\equiv_i$ on $\{0,1\}^*$ by

$$x \equiv_i y \iff f_i(x) = f_i(y).$$

Then the collision set C_{f_i} and the modified collision set C_i^* each consist of all but one of the elements of all the nonsingleton equivalence classes of $\equiv_i$. It follows immediately that C_{f_i} and C_i^* are either both finite or both infinite. □

Overview of the construction. Informally and intuitively, the language H is constructed by deciding the Boolean values $[\![\langle \mathbf{k}, \mathbf{n}\rangle \in H]\!]$ for successive (k,n) in the ordering $<^*$ of $\mathbf{N}^2$. It is convenient to regard H as consisting of the separate "strands" $H_k = \{\, \mathbf{n} \mid \langle \mathbf{k}, \mathbf{n}\rangle \in H \,\}$ for $k = 0, 1, 2, \ldots$. (See Fig. 3.) The construction exploits the ordering $<^*$ to ensure that $H \in \mathrm{E}_2$ and each $H_k \in \mathrm{E}$. The "ultimate objective" of each H_k is to ensure that a specially constructed martingale $\tilde{d}_k$ does not succeed on H_k. For each k, all but finitely many of the values $[\![\mathbf{n} \in H_k]\!]$ are chosen according to this ultimate objective. The exceptions occur

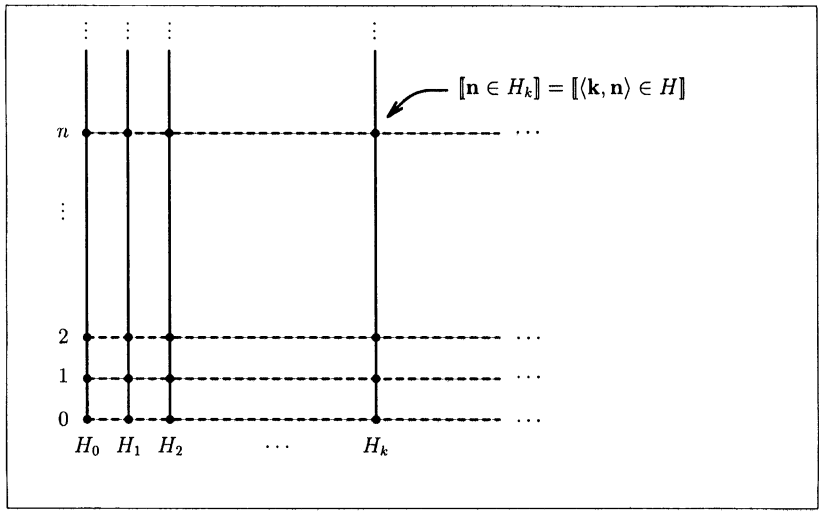


FIG. 3. *The strands of H.*

when values $[\![\mathbf{n} \in H_k]\!]$ are chosen to "destroy" various functions $f_i \in \mathrm{DTIMEF}(2^{4n})$, i.e., to ensure that these functions are *not* many-one reductions of H.

The specially constructed martingales are of the form $\widetilde{d}_k = d_k + \sum_{i=0}^{\infty} d_{i,k}$, where d_k is taken from the rigid enumeration of all p-martingales given by Theorem 3.4 and the martingales $d_{i,k}$ are defined below. Since $\widetilde{d}_k$ does not succeed on H_k, d_k also does not succeed on H_k. Since k is arbitrary here and each $H_k \in \mathrm{P}_m(H) \cap \mathrm{E}$, it follows that $\mathrm{P}_m(H) \cap \mathrm{E}$ does not have p-measure 0, i.e., that $\mathrm{P}_m(H)$ does not have measure 0 in E. Thus H is weakly $\leq_m^{\mathrm{P}}$-hard for E. On the other hand, since $\widetilde{d}_k$ does not succeed on H_k, none of the martingales $d_{i,k}$ succeeds on H_k. Moreover, matters are arranged so that, for every many-one reduction f_i of H with C_i^* infinite, either some $d_{i,k}$ succeeds on H_k or else f_i is eventually "destroyed" by some value $[\![\mathbf{n} \in H_k]\!]$. It follows that H is incompressible by $\leq_m^{\mathrm{DTIME}(2^{4n})}$-reductions, whence H is not $\leq_m^{\mathrm{P}}$-hard for E by Theorem 4.2.

Precise details follow.

The construction. The language $H \subseteq \{0, 1\}^*$ is defined by

$$H = \{ \langle \mathbf{k}, \mathbf{n} \rangle \mid \mathbf{n} \in H_k \},$$

where the languages $H_0, H_1, \ldots$ are defined, along with the auxiliary martingales $\widetilde{d}_0, \widetilde{d}_1, \ldots$, by the following recursion: (Recall that $d_0, d_1, \ldots; \widehat{d}_0, \widehat{d}_1, \ldots$ is a fixed rigid enumeration of all p-martingales.)

(1) For $k \in \mathbf{N}$ and $w \in \{0, 1\}^*$, define

$$\widetilde{d}_k(w) = d_k(w) + \sum_{i=0}^{\infty} d_{i,k}(w),$$

where the functions $d_{i,k}$ are computed as follows. Assume that $w \in \{0, 1\}^*$, $n = |w|$, and $b \in \{0, 1\}$.

(a) $d_{i,k}(\lambda) = 2^{-i}$.

(b) If $(k, n) \notin C_i^*$, then $d_{i,k}(wb) = d_{i,k}(w)$.

(c) If $(k, n) \in C_i^*$, then

$$d_{i,k}(wb) = 2 \cdot d_{i,k}(w) \cdot [\![b = [\![\mathbf{m} \in H_j]\!]]\!],$$

where (j, m) is the $<^*$-least pair in $\mathbf{N}^2$ such that $f_i(\langle \mathbf{j}, \mathbf{m} \rangle) = f_i(\langle \mathbf{k}, \mathbf{n} \rangle)$.
It is clear that each $d_{i,k}$, and hence each $\widetilde{d}_k$, is a martingale.
For $k, r \in \mathbf{N}$ and $w \in \{0, 1\}^*$, the approximation

$$\widehat{\widetilde{d}}_{k,r}(w) = \widehat{d}_{k,r+1}(w) + \sum_{i=0}^{r+|w|+1} d_{i,k}(w)$$

of $\widetilde{d}_k(w)$ is also used. It is easy to check that

$$\left| \widehat{\widetilde{d}}_{k,r}(w) - \widetilde{d}_k(w) \right| \leq 2^{-r}$$

for all $k, r \in \mathbf{N}$ and $w \in \{0, 1\}^*$ satisfying $r + 1 \geq |w|$.

(2) In the construction of the languages $H_0, H_1, \ldots$, the operation

$$\text{destroy } f_i \text{ at } (k, n)$$

is often performed. In all such instances, it is known that $(k, n) \in C_i^*$ and the operation is performed by setting

$$[\![\mathbf{n} \in H_k]\!] = [\![\mathbf{m} \notin H_j]\!],$$

where (j, m) is the $<^*$-least pair in $\mathbf{N}^2$ such that $f_i(\langle \mathbf{j}, \mathbf{m} \rangle) = f_i(\langle \mathbf{k}, \mathbf{n} \rangle)$. Note that a single performance of this operation ensures that f_i is not a many-one reduction of H.

The sets

$$D_{k,n} = \left\{ i \in \mathbf{N} \,\middle|\, (\exists (j, m) <^* (k, n))\ f_i \text{ is destroyed at } (j, m) \right\}$$

for $k, n \in \mathbf{N}$, are also used in the construction. It is emphasized that an index i appears in $D_{k,n}$ only if the operation destroy f_i at (j, m) is *explicitly* performed for some $(j, m) <^* (k, n)$. In particular, for each (j, m) there is at most one i such that f_i is destroyed at (j, m), even though there are many i' such that $f_i = f_{i'}$. Thus each $D_{k,n}$ is a finite set with $|D_{k,n}| \leq \#^*(k, n)$.

For $k, n \in \mathbf{N}$, let

$$\iota(k, n) = \min \left\{ i \in \mathbf{N} \,\middle|\, i \notin D_{k,n} \text{ and } (k, n) \in C_i^* \right\}.$$

Note that $\iota(k, n)$ is finite for all $k, n \in \mathbf{N}$ (because f_i is constant for infinitely many i). The values $[\![\mathbf{n} \in H_k]\!]$ are defined according to the following two cases.

Case 1. If $\iota(k, n) \leq k$, then destroy $f_{\iota(k,n)}$ at (k, n).

Case 2. If $\iota(k, n) > k$, then set

$$[\![\mathbf{n} \in H_k]\!] = \left[\!\left[\widehat{\widetilde{d}}_{k,n}(w1) \leq \widehat{\widetilde{d}}_{k,n}(w0) \right]\!\right],$$

```
begin
  w := χ_{H_k}[0..n − 1];
  for b ∈ {0, 1} do
  begin
    δ_b := d̂_{k,n+1}(wb)
    for i := 0 to 2n + 1 do δ_b := δ_b + d_{i,k}(wb)
  end; //Now δ_0 = d̃̃_{k,n}(w0) and δ_1 = d̃̃_{k,n}(w1).//
  if ι(k, n) ≤ k
    then destroy f_{ι(k,n)} at (k, n)
    else [[n ∈ H_k]] := [[δ_1 ≤ δ_0]]
end.
```

FIG. 4. *Computation of* $[\![\mathbf{n} \in H_k]\!]$ *in the proof of Lemma* 4.4.

where $w = \chi_{H_k}[0..n-1]$. This completes the construction of the languages $H_0, H_1, \ldots$ and the martingales $\widetilde{d}_0, \widetilde{d}_1, \ldots$.

The following lemmas are used to prove the main theorem.

LEMMA 4.4. $H \in \mathrm{E}_2$. *For each* $k \in \mathbf{N}$, $H_k \in \mathrm{E}$.

Proof. Assume first that $(k, n) \in \mathbf{N}^2$ and the values $[\![\mathbf{m} \in H_j]\!]$ are known (stored) for all pairs $(j, m) <^* (k, n)$, as is the set $D_{k,n}$. Consider the computation of $[\![\mathbf{n} \in H_k]\!]$ exhibited in Fig. 4.

To estimate the time required for this computation, recall the properties

$$(j, m) <^* (k, n) \Longrightarrow (1 + |\mathbf{j}|)(1 + |\mathbf{m}|) \leq (1 + |\mathbf{k}|)(1 + |\mathbf{n}|),$$

$$\#^*(k, n) = 2^{O((1+|\mathbf{k}|)(1+|\mathbf{n}|))}$$

of $<^*$ and note the following:

(i) The computation of w requires at most $n \cdot \#^*(k, n) = 2^{O((1+|\mathbf{k}|)(1+|\mathbf{n}|))}$ steps.

(ii) The computation of $\widehat{d}_{k,n+1}(wb)$ requires at most $(3 + n + |wb|)^{|\mathbf{k}|} = 2^{O((1+|\mathbf{k}|)(1+|\mathbf{n}|))}$ steps.

(iii) For $0 \leq i \leq 2n+1$, the condition $(k, n) \in C_i^*$ can be tested in at most $(1+\#^*(k, n))^2 \cdot O(2^{5|\langle \mathbf{i}, \langle \mathbf{k}, \mathbf{n} \rangle \rangle|}) = 2^{O(|\mathbf{i}|+O((1+|\mathbf{k}|)(1+|\mathbf{n}|)))} = 2^{O((1+|\mathbf{k}|)(1+|\mathbf{n}|))}$ steps.

(iv) By (iii), for $0 \leq i \leq 2n + 1$, the computation of $d_{i,k}(wb)$ requires at most $O(n \cdot 2^{O((1+|\mathbf{k}|)(1+|\mathbf{n}|))} \cdot 2^{O((1+|\mathbf{k}|)(1+|\mathbf{n}|))}) = 2^{O((1+|\mathbf{k}|)(1+|\mathbf{n}|))}$ steps.

(v) By (ii) and (iv), the entire computation of $\delta_b = \widetilde{\widetilde{d}}_{k,n}(wb)$, i.e., the for-loop in Fig. 4, requires at most $2^{O((1+|\mathbf{k}|)(1+|\mathbf{n}|))} + (2n + 2)2^{O((1+|\mathbf{k}|)(1+|\mathbf{n}|))} = 2^{O((1+|\mathbf{k}|)(1+|\mathbf{n}|))}$ steps.

(vi) As in (iii), for $0 \leq i \leq k$, the condition $(k, n) \in C_i^*$ can be tested in at most $2^{O((1+|\mathbf{k}|)(1+|\mathbf{n}|))}$ steps. Thus, testing the condition $\iota(k, n) \leq k$ and computing $\iota(k, n)$ if this condition is true requires at most $(k + 1) \cdot 2^{O((1+|\mathbf{k}|)(1+|\mathbf{n}|))} = 2^{O((1+|\mathbf{k}|)(1+|\mathbf{n}|))}$ steps. It follows easily that the if-then-else in Fig. 4 requires at most $2^{O((1+|\mathbf{k}|)(1+|\mathbf{n}|))}$ steps.

By (i), (v), and (vi), the computation described in Fig. 4 requires at most $2^{O((1+|\mathbf{k}|)(1+|\mathbf{n}|))}$ steps to compute $[\![\mathbf{n} \in H_k]\!]$, given the set $D_{k,n}$ and the values $[\![\mathbf{m} \in H_j]\!]$ for $(j, m) <^* (k, n)$.

The condition $\langle \mathbf{k}, \mathbf{n} \rangle \in H$ can now be decided by computing and storing the successive values $[\![\mathbf{m} \in H_j]\!]$ according to the $<^*$-ordering of $\mathbf{N}^2$, using the computation in Fig. 4 and updating $D_{j,n}$ at each stage. This requires at most $(1 + \#^*(k, n)) \cdot O(2^{O((1+|\mathbf{k}|)(1+|\mathbf{n}|))}) =$

$2^{O((1+|\mathbf{k}|)(1+|\mathbf{n}|))}$ steps. Since $2^{O((1+|\mathbf{k}|)(1+|\mathbf{n}|))} = 2^{O((1+|\langle \mathbf{k},\mathbf{n}\rangle|)^2)}$, this proves that $H \in \mathrm{E}_2$. Also, for fixed k, $2^{O((1+|\mathbf{k}|)(1+|\mathbf{n}|))} = 2^{O(1+|\mathbf{n}|)}$, so each $H_k \in \mathrm{E}$. □

LEMMA 4.5. *For all $i \in \mathbf{N}$, if there exist infinitely many $k \in \mathbf{N}$ such that the slice $C^*_{i,k}$ is nonempty, then f_i is not a many-one reduction of H.*

Proof. Fix $i \in \mathbf{N}$ and assume that the set

$$S = \left\{ k \in \mathbf{N} \,\middle|\, C^*_{i,k} \neq \emptyset \right\}$$

is infinite. For each $k \in S$, let $n_k = \min C^*_{i,k}$. For every $k \in S$, at least one of the following four conditions must hold:

(i) $i > k$.
(ii) $i < \iota(k, n_k)$.
(iii) $\iota(k, n_k) < i \leq k$.
(iv) $\iota(k, n_k) = i \leq k$.

(In fact, for all real numbers a, b, and c, at least one of $a > c$, $a < b$, $b < a \leq c$, $b = a \leq c$ must hold.) It is clear that condition (i) holds for only finitely many k. For each k such that condition (iii) holds, the construction of H ensures that $f_{\iota(k,n_k)}$ is destroyed at (k, n_k). Since each f_j is destroyed at most once in the construction of H, it follows that condition (iii) holds for only finitely many k. Since S is infinite, this implies that there is some $k \in S$ such that condition (ii) or condition (iv) holds.

Fix such a number k. If condition (ii) holds, then $i \in D_{k,n_k}$ (because $(k, n_k) \in C^*_i$), so f_i is not a many-one reduction of H. If condition (iv) holds, then f_i is destroyed at (k, n_k), so f_i is not a many-one reduction of H. Thus, in any case, f_i is not a many-one reduction of H. □

LEMMA 4.6. *For all $i, k \in \mathbf{N}$, if f_i is a many-one reduction of H and $C^*_{i,k}$ is infinite, then $d_{i,k}$ succeeds on H_k.*

Proof. Assume that $i, k \in \mathbf{N}$, f_i is a many-one reduction of H, and $C^*_{i,k}$ is infinite. Consider the successive values

$$r_n = d_{i,k}(\chi_{H_k}[0..n-1])$$

for $n = 0, 1, 2, \ldots$. Clause (a) of the definition of $d_{i,k}$ says that $r_0 = 2^{-i}$, while clauses (b) and (c) ensure that each $r_{n+1} \in \{0, r_n, 2r_n\}$. In fact, since f_i is a reduction of H, clause (c) never causes r_{n+1} to be 0. We thus have the recurrence

$$r_0 = 2^{-i}, \quad r_{n+1} = \begin{cases} r_n & \text{if } n \notin C^*_{i,k}, \\ 2r_n & \text{if } n \in C^*_{i,k}. \end{cases}$$

Since $C^*_{i,k}$ is infinite, this implies that $r_n \to \infty$ as $n \to \infty$, whence $d_{i,k}$ succeeds on H_k. □

LEMMA 4.7. *For all $k \in \mathbf{N}$, $\widetilde{d}_k$ does not succeed on H_k.*

Proof. Fix $k \in \mathbf{N}$ and consider the manner in which the values $[\![\mathbf{n} \in H_k]\!]$ are decided for $n = 0, 1, 2, \ldots$. There can be at most finitely many values of n for which Case 1 holds. (This is because each occurrence of Case 1 involves a new value of $\iota(k, n)$ with $\iota(k, n) \leq k$.) Thus there exists $n_0 \in \mathbf{N}$ such that Case 2 holds for all $n \geq n_0$. For all $n \in \mathbf{N}$, let

$$w_n = \chi_{H_k}[0..n-1]$$

be the n-bit prefix of χ_{H_k}. Then, for all $m \geq n_0$, Case 2 ensures that

$$\begin{aligned}
\widetilde{d}_k(w_{m+1}) &\leq \widehat{\widetilde{d}}_{k,m}(w_{m+1}) + 2^{-m} \\
&\leq \frac{\widehat{\widetilde{d}}_{k,m}(w_m 0) + \widehat{\widetilde{d}}_{k,m}(w_m 1)}{2} + 2^{-m} \\
&\leq \frac{\widetilde{d}_k(w_m 0) + 2^{-m} + \widetilde{d}_k(w_m 1) + 2^{-m}}{2} + 2^{-m} \\
&= \frac{\widetilde{d}_k(w_m 0) + \widetilde{d}_k(w_m 1)}{2} + 2^{1-m} \\
&= \widetilde{d}_k(w_m) + 2^{1-m}.
\end{aligned}$$

It follows that, for all $n \geq n_0$,

$$\widetilde{d}_k(w_n) \leq \widetilde{d}_k(w_{n_0}) + \sum_{m=n_0}^{n-1} 2^{1-m} < \widetilde{d}_k(w_{n_0}) + 4.$$

Thus, if

$$\sigma = \max_{0 \leq n \leq n_0} \widetilde{d}_k(w_n),$$

then

$$\widetilde{d}_k(w_n) < \sigma + 4$$

for all $n \in \mathbf{N}$. Hence $\widetilde{d}_k$ does not succeed on H_k. □

LEMMA 4.8. *H is weakly $\leq_m^{\mathrm{P}}$-hard for* E.

Proof. Let $k \in \mathbf{N}$. It is clear that $H_k \in \mathrm{P}_m(H)$ and $S^\infty[d_k] \subseteq S^\infty[\widetilde{d}_k]$. It follows by Lemmas 4.4 and 4.7 that

$$H_k \in \mathrm{P}_m(H) \cap \mathrm{E} - S^\infty[\widetilde{d}_k] \subseteq \mathrm{P}_m(H) \cap \mathrm{E} - S^\infty[d_k],$$

whence $\mathrm{P}_m(H) \cap \mathrm{E} \not\subseteq S^\infty[d_k]$. Since k is arbitrary here, this implies that $\mu_{\mathrm{p}}(\mathrm{P}_m(H) \cap \mathrm{E}) \neq 0$, i.e., $\mu(\mathrm{P}_m(H) \mid \mathrm{E}) \neq 0$. Thus H is weakly $\leq_m^{\mathrm{P}}$-hard for E. □

LEMMA 4.9. *H is not $\leq_m^{\mathrm{P}}$-hard for* E.

Proof. By Theorem 4.2, it suffices to show that H is incompressible by $\leq_m^{\mathrm{DTIME}(2^{4n})}$-reductions. Fix $i \in \mathbf{N}$ such that f_i is a many-one reduction of H. It suffices to show that f_i is one-to-one almost everywhere.

Note the following two things:

(i) For each $k \in \mathbf{N}$, the slice $C_{i,k}^*$ is finite by Lemmas 4.7 and 4.6.

(ii) By Lemma 4.5, there are only finitely many $k \in \mathbf{N}$ such that $C_{i,k}^* \neq \emptyset$.

Taken together, (i) and (ii) imply that C_i^* is finite. It follows by Lemma 4.3 that f_i is one-to-one almost everywhere. □

By Lemmas 4.4, 4.8, and 4.9, the language $H \in \mathrm{E}_2$ is weakly $\leq_m^{\mathrm{P}}$-hard but not $\leq_m^{\mathrm{P}}$-hard for E. From this, a simple padding argument suffices to prove the main theorem.

Proof of main theorem. Let H be defined as above. By Lemma 4.4, there is a polynomial $q(n) \geq n$ such that $H \in \mathrm{DTIME}(2^{q(n)})$. Let

$$C = \left\{ x10^{q(|x|)} \mid x \in H \right\}.$$

It is easy to check that $C \in \mathrm{E}$ and $\mathrm{P}_m(C) = \mathrm{P}_m(H)$. It follows by Lemmas 4.8 and 4.9 that C is weakly $\leq_m^{\mathrm{P}}$-complete but not $\leq_m^{\mathrm{P}}$-complete for E. □

5. Conclusion. The most important problem suggested by this work is the finding of "natural" examples of languages that are weakly $\leq_m^{\mathrm{P}}$-complete but not $\leq_m^{\mathrm{P}}$-complete for E. As noted in §1, such languages would provably be strongly intractable. It is reasonable to hope that the study of such natural examples would yield new insights into the nature of intractability.

It is especially intriguing to consider the possibility that SAT and other natural NP-complete problems may be weakly $\leq_m^{\mathrm{P}}$-complete for E, i.e., that NP may not have measure 0 in E. The hypothesis that SAT is weakly $\leq_m^{\mathrm{P}}$-complete for E implies, but may in some sense be stronger than, the $\mathrm{P} \neq \mathrm{NP}$ hypothesis. For example, recent work has shown that, if SAT is weakly $\leq_m^{\mathrm{P}}$-complete for E, then NP contains P-bi-immune languages [14], every $\leq_m^{\mathrm{P}}$-hard language for NP is dense [12], every $\leq_m^{\mathrm{P}}$-complete language for NP has a dense exponential complexity core [7], and there is a language that is $\leq_{\mathrm{T}}^{\mathrm{P}}$-complete but not $\leq_m^{\mathrm{P}}$-complete for NP [11]. Further investigation of the consequences and reasonableness of this hypothesis is indicated.

It is routine to modify the proof of the main theorem to construct languages that are weakly $\leq_m^{\mathrm{P}}$-complete but not $\leq_m^{\mathrm{P}}$-complete for larger classes such as E_2 and ESPACE. A more interesting, and perhaps harder question concerns alternate versions of the main theorem in which $\leq_m^{\mathrm{P}}$ is replaced by other reducibilities. Homer, Kurtz, and Royer [5] have proven that a language is $\leq_{1-tt}^{\mathrm{P}}$-hard for E if and only if it is $\leq_m^{\mathrm{P}}$-hard for E. It follows immediately that the language C given by the main theorem is weakly $\leq_{1-tt}^{\mathrm{P}}$-complete but not $\leq_{1-tt}^{\mathrm{P}}$-complete for E. That is, the main theorem holds with $\leq_m^{\mathrm{P}}$ replaced by $\leq_{1-tt}^{\mathrm{P}}$. Beyond this, little is known. New techniques may be required to determine whether the main theorem holds with $\leq_m^{\mathrm{P}}$ replaced by $\leq_{\mathrm{T}}^{\mathrm{P}}$.

Acknowledgments. I thank Jim Royer and Tom Linton for several useful suggestions on an earlier draft of this paper. I also thank two anonymous referees for several corrections and helpful suggestions.

REFERENCES

[1] L. Berman and J. Hartmanis, *On isomorphism and density of NP and other complete sets*, SIAM J. Comput., 6 (1977), pp. 305-322.

[2] P. R. Halmos, *Measure Theory*, Springer-Verlag, Berlin, New York, Heidelberg, 1950.

[3] J. Hartmanis and R. E. Stearns, *On the computational complexity of algorithms*, Trans. Amer. Math. Soc., 117 (1965), pp. 285-306.

[4] F. C. Hennie and R. E. Stearns, *Two-tape simulation of multitape Turing machines*, J. Assoc. Comput. Mach., 13 (1966), pp. 533-546.

[5] S. Homer, S. Kurtz, and J. Royer, *On 1-truth-table-hard languages*, Theoret. Comput. Sci., 115 (1993), pp. 383-389.

[6] D. T. Huynh, *Some observations about the randomness of hard problems*, SIAM J. Comput., 15 (1986), pp. 1101-1105.

[7] D. W. Juedes and J. H. Lutz, *The complexity and distribution of hard problems*, SIAM J. Comput., 24 (1995), pp. 279-295; Proc 34th IEEE Symposium on Foundations of Computer Science, Palo Alto, CA, 1993, pp. 177-185.

[8] J. H. LUTZ, *Category and measure in complexity classes*, SIAM J. Comput., 19 (1990), pp. 1100-1131.
[9] ———, *Almost everywhere high nonuniform complexity*, J. Comput. System Sci., 44 (1992), pp. 220-258.
[10] ———, *Resource-bounded measure*, in preparation.
[11] J. H. LUTZ AND E. MAYORDOMO, *Cook versus Karp-Levin: Separating completeness notions if NP is not small*, Theoret. Comput. Sci., to appear; Proc. 11th Annual Symposium on Theoretical Aspects of Computer Science, Springer-Verlag, Berlin, New York, Heidelberg, 1994, pp. 415-426.
[12] ———, *Measure, stochasticity, and the density of hard languages*, SIAM J. Comput., 23 (1994), pp. 762-779.
[13] N. LYNCH, *On reducibility to complex or sparse sets*, J. Assoc. Comput. Mach., 22 (1975), pp. 341-345.
[14] E. MAYORDOMO, *Almost every set in exponential time is P-bi-immune*, Theoret. Comput. Sci., 136 (1994), pp. 487-506; Proc. 17th International Symposium on Mathematical Foundations of Computer Science, Springer-Verlag, New York, Berlin, Heidelberg, 1992, pp. 392-400.
[15] A. R. MEYER, 1977, reported in [1].
[16] P. ORPONEN AND U. SCHÖNING, *The density and complexity of polynomial cores for intractable sets*, Inform. and Control, 70 (1986), pp. 54-68.
[17] J. C. OXTOBY, *Measure and Category*, 2nd ed., Springer-Verlag, New York, Berlin, Heidelberg, 1980.
[18] H. L. ROYDEN, *Real Analysis*, 2nd ed., Macmillan, New York, 1968.
[19] C. P. SCHNORR, *Klassifikation der Zufallsgesetze nach Komplexität und Ordnung*, Z. Wahrscheinlichkeitstheorie verw. Geb., 16 (1970), pp. 1-21.
[20] ———, *A unified approach to the definition of random sequences*, Math. Systems Theory, 5 (1971), pp. 246-258.
[21] ———, *Zufälligkeit und Wahrscheinlichkeit*, Lecture Notes in Mathematics 218, Springer-Verlag, Berlin, Heidelberg, 1971.
[22] ———, *Process complexity and effective random tests*, J. Comput. System Sci., 7 (1973), pp. 376-388.
[23] U. SCHÖNING, *Complete sets and closeness to complexity classes*, Mathematical Systems Theory, 19 (1986), pp. 29-41.
[24] L. STOCKMEYER AND A. K. CHANDRA, *Provably difficult combinatorial games*, SIAM J. Comput., 8 (1979), pp. 151-174.
[25] L. J. STOCKMEYER, *Classifying the computational complexity of problems*, J. Symbolic Logic, 52 (1987), pp. 1-43.

SIAM J. COMPUT.
Vol. 24, No. 6, pp. 1190–1206, December 1995

DATA-STRUCTURAL BOOTSTRAPPING, LINEAR PATH COMPRESSION, AND CATENABLE HEAP-ORDERED DOUBLE-ENDED QUEUES*

ADAM L. BUCHSBAUM†, RAJAMANI SUNDAR‡, AND ROBERT E. TARJAN§

Abstract. A *deque with heap order* is a linear list of elements with real-valued keys that allows insertions and deletions of elements at both ends of the list. It also allows the *findmin* (alternatively *findmax*) operation, which returns the element of least (greatest) key, but it does not allow a general *deletemin* (*deletemax*) operation. Such a data structure is also called a *mindeque* (*maxdeque*). Whereas implementing heap-ordered deques in constant time per operation is a solved problem, catenating heap-ordered deques in sublogarithmic time has remained open until now.

This paper provides an efficient implementation of catenable heap-ordered deques, yielding constant amortized time per operation. The important algorithmic technique employed is an idea that we call *data-structural bootstrapping*: we abstract heap-ordered deques by representing them by their minimum elements, thereby reducing catenation to simple insertion. The efficiency of the resulting data structure depends upon the complexity of a special case of path compression that we prove takes linear time.

Key words. path compression, lists, queues, deques, catenation, heap order, priority queues, data-structural bootstrapping

AMS subject classifications. 68P05, 68Q25, 68R05

1. Introduction.

A *deque with heap order* is a linear list of elements with real-valued keys that allows insertions and deletions of elements at both ends of the list. It also allows the *findmin* (alternatively *findmax*) operation, which returns the element of least (greatest) key, but it does not allow a general *deletemin* (*deletemax*) operation. Such a data structure is also called a *mindeque* (*maxdeque*). The restricted access and lack of deletemin distinguish heap-ordered deques from general heaps and allow faster operation times than heaps. Gajewska and Tarjan [11] show how to implement heap-ordered deques with constant time (amortized or worst-case) per operation; they leave open the problem of how to catenate heap-ordered deques.

This paper provides an efficient implementation of catenable heap-ordered deques; we achieve constant amortized time per operation. The important algorithmic technique employed is an idea of Driscoll, Sleator, and Tarjan [8], which we call *data-structural bootstrapping*: we abstract the heap-ordered deques of Gajewska and Tarjan by representing them in terms of their minimum elements; in this way, catenation becomes no more difficult than insertion. To prove that the resulting data structure achieves constant amortized time per operation, we consider *order-preserving path compression*. This is a generalization of special cases of path compression originally introduced by Hart and Sharir [12] and subsequently analyzed by Loebl

*Received by the editors December 28, 1992; accepted for publication (in revised form) June 3, 1994. A preliminary version of this paper was presented at the 33rd IEEE Symposium on Foundations of Computer Science, 1992.

†AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, New Jersey 07974 (alb@research.att.com). The research of this author was performed at Princeton University and supported by a Fannie and John Hertz Foundation fellowship, National Science Foundation grant CCR-8920505, and the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) under grant NSF-STC88-09648.

‡Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560012, India (sundar@chanakya.csa.iisc.ernet.in). The research of this author was performed at Princeton University and the Center for Discrete Mathematics and Theoretical Computer Science and was supported by the Center for Discrete Mathematics and Theoretical Computer Science under grant NSF-STC88-09648.

§Department of Computer Science, Princeton University, Princeton, New Jersey 08544 and NEC Research Institute, 4 Independence Way, Princeton, New Jersey 08540 (ret@cs.princeton.edu). The research of this author at Princeton University was partially supported by National Science Foundation grant CCR-8920505, Office of Naval Research contract N00014-91-J-1463, and the Center for Discrete Mathematics and Theoretical Computer Science under grant NSF-STC88-09648.

and Nešetřil [17]–[19] and Lucas [20]. We prove a linear bound on *deque-ordered spine-only path compression*, a case of order-preserving path compression used in our data structure.

Our result is important in the following respects. It shows how the bootstrapping technique of Driscoll, Sleator, and Tarjan, originally developed to create confluently persistent catenable lists, is in fact a generally useful tool in the design of efficient data structures. Additionally, we not only unify the special cases of path compression considered by Loebl and Nešetřil and by Lucas, but we extend their results to a more general case. Furthermore, we provide what we believe is the first practical application of this type of result.

Sections 2 and 3 of this paper describe our data structure for catenable heap-ordered deques, using the bootstrapping technique of Driscoll, Sleator, and Tarjan. Sections 4 and 5 define and analyze deque-ordered spine-only path compression, proving the linearity of this special case of path compression. These latter sections are the technically difficult part of the paper. In §6, we prove a nonlinear lower bound for general order-preserving path compression and provide a nontrivial worst-case bound per heap-ordered deque operation based on an alternative implementation. We conclude in §7. A preliminary version of this paper appears as Buchsbaum, Sundar, and Tarjan [4].

2. Lists, heap order, and catenation. Consider the following operations to be performed on a linear list d of elements:

push(x, d) Insert x as the new first element of d; the previous ith element becomes the $(i+1)$st.

pop(d) Remove and return the first element of d (or $\emptyset$ if d is empty); the previous ith element, for $i \geq 2$, becomes the $(i-1)$st.

inject(x, d) Insert x as the new last element of d.

eject(d) Remove and return the last element of d (or $\emptyset$ if d is empty).

We assume the existence of a *makelist* operation that returns an initially empty list. If only push and pop (or inject and eject) are allowed, d is a *stack* (formally a *list of type stack*). If only push and eject (or inject and pop) are allowed, d is a *queue*. If all the operations are allowed, d is a *double-ended queue* or *deque*. If both insertion operations but only one of the deletion operations are allowed, d is an *output-restricted deque*. Such data structures can easily be implemented by doubly linked (in some cases singly linked) lists yielding $O(1)$ worst-case times for each of the allowed operations [23].

If each element in d has a real-valued *key*, we may also want to consider the following operation:

findmin(d) Find and return an element of minimum key in d (or $\emptyset$ if d is empty).

Findmin does not modify the list itself. If d is a deque, then d together with findmin is a *heap-ordered deque* or *mindeque*. Analogous data structures are obtained by adding findmin to stacks, queues, and output-restricted queues. A related data structure is the *priority queue with attrition* [22]. We can also consider the *findmax* operation but restrict ourselves without loss of generality to findmin for the remainder of this paper.

Finally, if d_1 and d_2 are lists of the same type and of size (number of elements) s_1 and s_2, respectively, then we define the following operation:

catenate(d_1, d_2) Add the elements of d_2 to the back of d_1; i.e., let the (s_1+i)th element of d_1 be the ith element of d_2 for $1 \leq i \leq s_2$. The first s_1 elements of d_1 are unchanged. This operation destroys d_2.

Catenating lists in $O(1)$ time is straightforward; catenating heap-ordered lists is more problematic. This paper demonstrates how to implement *catenable heap-ordered deques* (or *catenable mindeques*) efficiently. In particular, our data structure performs n insertions (pushes and injects), m deletions (pops and ejects), and q catenations, all intermixed on $q+1$ catenable heap-ordered deques in total time $O(n+m+q)$, such that findmin always takes

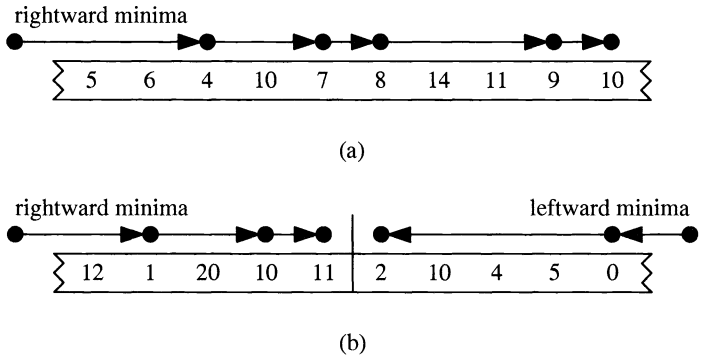


FIG. 2.1. (a) *A heap-ordered queue with minimum key* 4. (b) *A heap-ordered deque with minimum key* 0; *the center line is the break point between the two stacks.*

$O(1)$ worst-case time. Furthermore, if q is fixed, each operation requires $O(1)$ worst-case time.

We remark that an equivalent formulation of the problem is to have the makelist operation take an element as an argument and return a list of one element. Then push and inject can be treated as special cases of catenation. We choose the former suite of operations to distinguish catenation as a special operation that complicates the implementation of heap-ordered deques.

2.1. Related work and applications. Queues with heap order are useful in pagination [7], [13], [16], [21] and very large scale integration (VLSI) river routing [6]. Booth and Westbrook [3] use catenable heap-ordered queues in the sensitivity analysis of minimum spanning trees, shortest path trees, and minimum cost network flow on planar graphs. Larmore and Hirschberg [16] and Cole and Siegel [6] independently showed how to implement heap-ordered queues in $O(1)$ amortized time per operation [24]. Gajewska and Tarjan [11] modified their techniques to produce heap-ordered deques with $O(1)$ time per operation; they give both amortized and worst-case solutions. Applications of heap-ordered deques include computing all pairs shortest path information [10] and external farthest neighbors for simple polygons [1]. Whereas a simple extension of the previous heap-ordered queue techniques yields efficiently catenable heap-ordered queues (again $O(1)$ amortized time per operation), Gajewska and Tarjan left the question of how to construct efficiently catenable heap-ordered deques as an open problem. We address this problem.

We remark that in a preliminary version of this paper [4], we posed the designing of catenable heap-ordered deques with $O(1)$ worst-case time per operation (as opposed to our amortized time solution) as an open problem. Kosaraju [15] subsequently solved this problem. We contrast the two results in §7. (See also the note added in proof.)

2.2. Reviewing heap-ordered queues and deques. For completeness, we begin by reviewing how to implement heap-ordered queues and deques. Identify the head of a list (where the first element sits) as the left end of the list and the tail as the right. A *minimum element* of a list is an element of minimum key. To implement a heap-ordered queue d, maintain a secondary list d' of *rightward minima*. This list has as its first element the minimum element x of d; the second element is the minimum element of d to the right of x and so on; see Fig. 2.1(a).

To pop an element e from d, if e is minimum, the new minimum element of d becomes the successor of e in d' (and that element becomes the new head of d', i.e., d' is also popped). Otherwise, no change to d' takes place. To inject an element e of key k into d, consider

the rightmost element x of d'. If key$(x) > k$, eject x from d' and continue the search with the new rightmost element of d'. Once the rightmost element of d' is of lesser value than k, inject k into d'; note that k might be the only element of d' after this procedure. The findmin operation on d is now implemented by returning the head of d'. This is one possible implementation of the previous technique [6], [16] yielding $O(1)$ amortized time bounds for all the queue operations. We note that the technique also extends to the implementation of heap-ordered stacks, which can be used to create heap-ordered deques as well as heap-ordered output-restricted deques. Furthermore, it is simple to add catenation to heap-ordered queues, stacks, and output-restricted deques using the same idea.

Gajewska and Tarjan [11] implement heap-ordered deques by representing them as two heap-ordered stacks as in Fig. 2.1(b). When one stack is emptied, the heap-ordered deque is rebuilt so that the two stacks differ in size by no more than one. The findmin operation returns the minimum of the minimum elements in the left and right stacks. By gradually rebuilding the stacks concurrently with the deque operations, they achieve $O(1)$ worst-case time per operation. Using two stacks, however, does not allow the easy implementation of catenation that is possible with heap-ordered queues.

3. Data-structural bootstrapping and catenable heap-ordered deques. We employ a technique of Driscoll, Sleator, and Tarjan [8] to bootstrap the heap-ordered deques and allow their efficient catenation. We wish to implement a catenable heap-ordered deque d; call the elements of d *basic elements*. We freely interchange the notion of elements and their keys (e.g., $x > y$ for two elements x and y implies that the key of x is greater than the key of y).

Our data structure D is itself a heap-ordered deque. Each element of D consists of a basic element of d combined with a pointer. For an element x of D, let e be the basic element of x and p be the pointer of x. If $p = \emptyset$, then e is a basic element of d; in this case we call x a *d-element* of D. Otherwise, e is the minimum basic element of a heap-ordered deque to which p points. This other heap-ordered deque has the same type of elements as D, and the structure is recursive; the minimum element of D thus contains the minimum basic element among all those in the heap-ordered deques reachable via pointers from D. The d-elements of D occur in a natural *left-to-right* order that can be obtained by recursively visiting each element from head to tail in D, listing d-elements as they are encountered. The resulting ordered list of d-elements corresponds to the basic elements in order, in the catenable heap-ordered deque d we are implementing.

In this fashion, there is a one-to-one correspondence between the elements of D and the nodes of a heap-ordered tree; see Fig. 3.1(a)-(b). In the tree, the leaves represent d-elements of D and hence basic elements of d, and we maintain a *heap invariant* on the tree: the nonleaf (or internal) nodes contain the lowest values among their children. The internal nodes are precisely the non-d elements of D and the heap-ordered deques reachable from D. The leaves in the tree also have a natural left-to-right order corresponding to that on the d-elements of D.

The concept of representing a heap-ordered linear list of items by exploiting the induced left-to-right order of the leaves in a normal heap-ordered tree arises in the pagodas of Françon, Viennot, and Vuillemin [9]. Similar data structures are the Cartesian tree [27] and the treap [2]. These maintain one tree under both symmetric and heap orders (on two distinct keys per node). If the symmetrically ordered key represents the position of the node in a linear list, the data structure supports heap-ordered list access operations. The idea of bootstrapping heap-ordered deques to implement catenable heap-ordered deques in the above recursive fashion generalizes the technique of Kosaraju [14], by which he designs catenable deques (not heap ordered) by decomposing the deques into contiguous pieces and storing those pieces in a stack. His data structure can be extended to maintain heap order, but it only accommodates a fixed number of deques. Driscoll, Sleator, and Tarjan [8] bootstrap fully persistent lists to

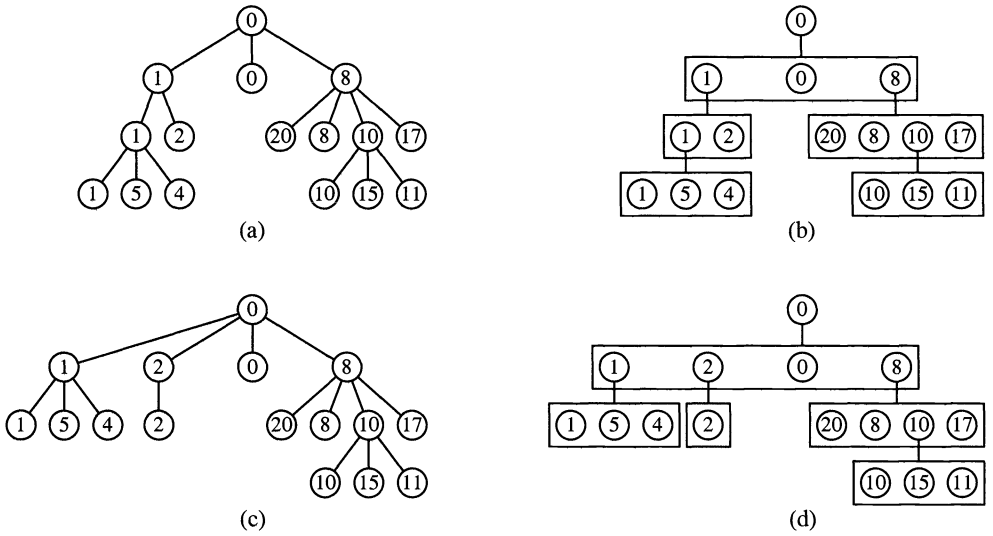


FIG. 3.1. (a) *A heap-ordered tree.* (b) *The corresponding catenable heap-ordered deque: boxes surround heap-ordered deques, the minimum values of which are elements in the parent heap-ordered deque.* (c) *The tree of (a) after a left pull.* (d) *The corresponding catenable heap-ordered deque.*

implement confluently persistent catenable lists. Recently, Buchsbaum and Tarjan [5] further refined bootstrapping into two distinct types, *structural abstraction*, which is the method of Driscoll, Sleator, and Tarjan, and *structural decomposition*, and use them to design efficient, confluently persistent deques.

3.1. The pull operation. We now define a *left pull* operation on the heap-ordered tree T in a way similar to the definition of the *pull* operation of Driscoll, Sleator, and Tarjan [8]. If the leftmost child x of the root of T is a leaf, a *left pull on* T does nothing. Otherwise, a left pull on T removes the leftmost child x' from x and makes x' a child of the root of T to the left of x; x' thus becomes the new leftmost child of the root of T, and if x is now a leaf, it is deleted. See Fig. 3.1(c) for an example. The heap invariant of the tree is maintained by updating x, if necessary, to contain the lowest key among those of its children. We define a *right pull on* T symmetrically. Both pull operations preserve the left-to-right order of the leaves of T.

A left pull is now extended to the data structure D as follows. If the first element x of D is a d-element (has a $\emptyset$ pointer), a left pull on D does nothing. Otherwise, pop D, returning x. Denote by $d(x)$ the heap-ordered deque to which the pointer of x points. Pop $d(x)$, returning x'. If $d(x)$ is still nonempty, push x back onto D; the pointer of x still points to $d(x)$, but popping $d(x)$ may change the basic element of x (the minimum basic element of $d(x)$). Then push x' onto D. A right pull on D is defined symmetrically. Since we use only heap-ordered deque operations to implement pulls on D, the minimum values of all the heap-ordered deques, in particular, that of D, are maintained correctly. The popping and repushing of x are executed precisely to maintain the correct minimum values; i.e., if x' is the minimum element of x, then popping x' from x and pushing it onto D necessitates changing the minima lists of D, which is effected by the popping and pushing of x before the pushing of x'. Figure 3.1(d) shows the result of a left pull on D. Again, the pull operations preserve the left-to-right order of the d-elements of D.

It is now straightforward to implement the catenable heap-ordered deque operations on d as follows: (We assume that the makelist operation returns an empty heap-ordered deque D.)

push(x, d)	Create element $\tilde{x}$ composed of basic element x and pointer $\emptyset$. Push $\tilde{x}$ onto D.
pop(d)	Repeatedly left pull D until its head element is a d element. Then pop D; this returns an element $\tilde{x}$. Return the basic element x of $\tilde{x}$.
inject(x, d)	Symmetric to push.
eject(d)	Symmetric to pop.
findmin(d)	Return the basic element in the minimum element of D.
catenate(d_1, d_2)	For $i \in \{1, 2\}$, let $\tilde{D}_i$ be an element containing the basic element in the minimum element of D_i and a pointer to D_i. Push $\tilde{D}_1$ onto D_2. (Alternatively, we could inject $\tilde{D}_2$ into D_1.)

That D correctly simulates d is easily proven by induction on the number of operations.

3.2. Time analysis. We now state the following theorem.

THEOREM 3.1. *An intermixed set of n insertions, m deletions, and q catenations on $q + 1$ catenable heap-ordered deques takes total time $O(n + m + q)$; findmin always takes $O(1)$ worst-case time.*

Proof. We first note that push (inject) requires a constant amount of time to prepare the element (from the given basic element and a $\emptyset$ pointer) in addition to one "real" heap-ordered deque push (inject). If we use the Gajewska–Tarjan heap-ordered deques [11], each real heap-ordered deque operation takes $O(1)$ time. Either the amortized or worst-case solution they offer suffices for the data structure in this section; the implementation given in §6.2 requires their worst-case solution. Continuing, each catenation also entails one real push or inject in addition to some constant amount of preprocessing time. Each deletion entails some number of pulls, each of which requires a constant number of real heap-ordered deque operations plus one final real pop or eject.

If we consider the corresponding heap-ordered tree, we see that the set of insertions, catenations, and deletions maps to an instance of disjoint set union [25]. In particular, the insertions and catenations correspond to unions and the deletions correspond to finds on the elements that are eventually deleted. That is, a sequence of pulls effects a path compression. Furthermore, note that the path compressions are all *spine compressions*. This notion will be defined in detail in §4; briefly, each compression involves a path of only leftmost children (or only rightmost children). Theorem 5.4 will prove the linearity of such path compressions. It is unnecessary to maintain the heap-ordered trees in a balanced fashion (e.g., doing unions by size or rank).

Findmin is performed by one real heap-ordered deque findmin, which takes $O(1)$ worst-case time. □

Note that if $q = O(1)$, that is, if there is only a constant number of heap-ordered deques to be catenated, our data structure implements all the operations in $O(1)$ time each. Thus our structure unifies the general problem with this special case mentioned by Gajewska and Tarjan [11], which they solved by using the techniques of Kosaraju [14].

COROLLARY 3.2. *An intermixed sequence of insertions, catenations, deletions, and findmins can be performed on a fixed number of catenable heap-ordered deques in $O(1)$ worst-case time per operation.*

Proof. If $q = O(1)$, each sequence of pulls is of constant length. □

4. Path compression. In this section we introduce some definitions necessary in the analysis of our special case of path compression. Initially, we have a tree T with n nodes (and $n - 1$ edges). An edge (u, v) connects node u with its *parent* v in the tree; we say that $p(u) = v$ *in* T. Each node, except for the root of the tree, has precisely one edge joining it to its parent. The parent pointers define a *path* from x to the root of T in the natural way. A node

x is a *descendant* of a node y if y lies on the path from x to the root of the tree; symmetrically, y is an *ancestor* of x. Note that x is both an ancestor and a descendant of itself. A *proper ancestor* (*descendant*) of x is an ancestor (descendant) y of x such that $y \neq x$.

A *path compression from* x_0 on a tree T is a sequence of nodes $C = (x_0, \ldots, x_l)$ such that $l > 0$ and $p(x_i) = x_{i+1}$ in T for $0 \leq i < l$. Its effect is to update the parents of the nodes along the path, making them all point to x_l, the *root of the compression*,

- $p(x_i) \leftarrow x_l,\ 0 \leq i < l$.

We say that the compression C *roots at* x_l; furthermore, x_i for $0 \leq i < l$ is a *nonroot node* of the path compression. The *cost* of C is $|C| = l$.

If $C = (x_0)$ and x_0 is a *leaf node*, i.e., a node with no children, then C is a *leaf deletion*; the effect of C is to remove x from T. In this case the cost of C is $|C| = 1$. A *sequence of path compressions on* $T = T_0$ is a sequence $(C_1, \ldots, C_m)$ such that C_i is a path compression or a leaf deletion on T_{i-1} and the result of C_i applied to T_{i-1} is T_i. The *cost* of a sequence of path compressions on T is $\sum_{i=1}^{m} |C_i|$.

We now define the *rising roots condition* [20], which links the notion of path compressions above with the well-known union and find operations used in the disjoint set union problem (see, e.g., Tarjan and van Leeuwen [25]). For any node x in T, let k_x be the smallest i such that x appears as a nonroot node in C_i; $k_x = \infty$ if there is no such i.

DEFINITION 4.1 (rising roots condition). *A sequence of path compressions* $(C_1, \ldots, C_m)$ *satisfies the* rising roots condition *if and only if for every node* x *and every* $i > k_x$, x *appears as a nonroot node in* C_i *if* C_i *is a compression from a descendant of* x *in* T_{i-1}.

The rising roots condition tells us precisely when a sequence of path compressions on some initial tree corresponds to an intermixed sequence of unions, finds, and leaf deletions.

LEMMA 4.2. *A sequence of path compressions (on an initial tree) satisfying the rising roots condition corresponds to some sequence of intermixed union, find, and leaf deletion operations. Conversely, a sequence of intermixed union, find, and leaf deletion operations corresponds to some sequence of path compressions satisfying the rising roots condition.*

Proof. See Lucas [20, Lem. 1]. □

The above correspondence is straightforward. The roots of the compressions in the path compression sequence are the roots of the finds in the union-find instance, and vice versa. This correspondence can be used to simplify the analysis of disjoint set union instances by assuming that all the unions are done before the first find. Any result for a class of path compressions satisfying the rising roots condition maps to a result for a class of disjoint set union problems.

We now introduce the notion of order to restrict the path compression sequences we shall consider. Given a tree T, embed T in the plane, yielding a left-to-right order on the children of each node. The *nearest common ancestor* of two nodes x and y ($\mathrm{nca}(x, y)$) is the deepest node z in T such that z is an ancestor of both x and y. For x, a proper descendant of z, let $c_x(z)$ be the child of z that is an ancestor of x; note that $c_x(z)$ might equal x. We define a partial order on T as follows: for any pair of nodes x and y, if $z = \mathrm{nca}(x, y)$ and $z \notin \{x, y\}$, then $x \lessdot y$ if $c_x(z)$ is to the left of $c_y(z)$.

DEFINITION 4.3 (order preservation). *A path compression on a tree* T *that yields a tree* T' *is* order preserving *if* $x \lessdot y$ *in* $T \Longrightarrow x \lessdot y$ *in* T'. *A leaf deletion is always taken as* order preserving. *A sequence* $(C_1, \ldots, C_m)$ *of path compressions on* T_0 *is* order preserving *if* C_i *is order preserving for* $1 \leq i \leq m$.

Note that as there is, in general, more than one way to effect a path compression (in terms of the left-to-right order of the newly acquired children of the root of the compression), Definition 4.3 depends upon the actual implementation of the path compressions involved.

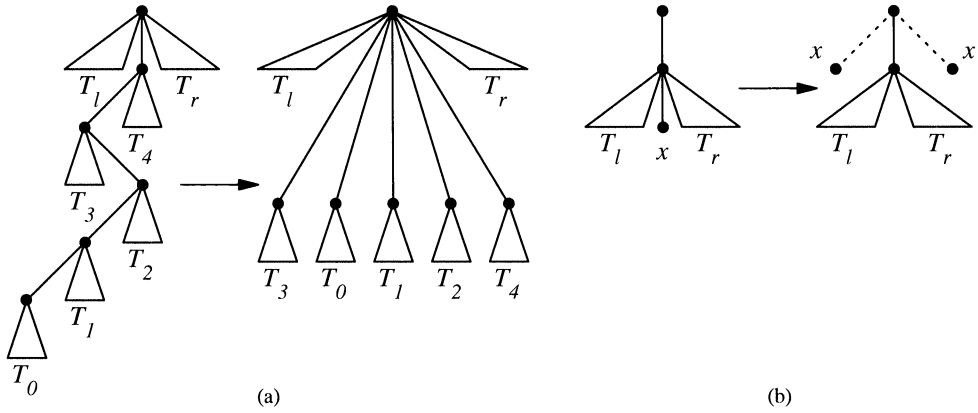


FIG. 4.1. (a) *An order-preserving path compression; triangles denote subtrees.* (b) *A path compression that cannot be order-preserving; x must become either left of T_l or right of T_r.*

We assume for simplicity that a compression is effected in an order-preserving way if it can be. We now describe exactly when a path compression (sequence) is order preserving.

LEMMA 4.4. *A path compression $(x_0, \ldots, x_l)$ is order preserving if and only if x_i is the leftmost or rightmost child of x_{i+1} for $0 \leq i < l - 1$.*

Proof. First note that the last nonroot node in the compression need not be an *extremal* (leftmost or rightmost) child of its parent; only the nodes whose parents change need to be extremal.

Now, assume that x_i is an extremal child of x_{i+1} for $0 \leq i < l - 1$. Then we can effect the path compression top-down in an order preserving way as follows. If x_{l-2} is the leftmost (rightmost) child of x_{l-1}, and y is the rightmost (leftmost) sibling to the left (right) of x_{l-1}, make x_{l-2} a new child of x_l in between y and x_{l-1}; y might be nil. Continue in this fashion down to x_0; see Fig. 4.1(a).

On the other hand, let $j < l - 1$ be such that x_j is not an extremal child of x_{j+1}. Let L be any left sibling of x_j and R be any right sibling of x_j. Before the compression we have $L \lessdot x_j \lessdot R$. If, however, x_j becomes a left sibling of x_{j+1} during the compression, then after the compression we have $x_j \lessdot L$; similarly, if x_j becomes a right sibling of x_{j+1}, then after the compression we have $R \lessdot x_j$. See Figure 4.1(b). □

Note that an only child, i.e., a node with no siblings, qualifies as both a leftmost and a rightmost child of its parent. Now consider a postorder (preorder) assigned to the nodes of a tree. In this paper we consider a restricted case of order-preserving path compression.

DEFINITION 4.5 (spine compression). *A path compression $(x_0, \ldots, x_l)$ is a* left-spine compression (right-spine compression) *if x_i is the leftmost (rightmost) child of x_{i+1} for $0 \leq i < l - 1$ and the compression preserves the postorder (preorder) of the tree. A sequence of path compressions is a sequence of* spine-only *path compressions if each path compression in the sequence is a left-spine compression or a right-spine compression.*

Clearly left- and right-spine compressions are order preserving. Lucas [20] proves that a sequence of left-spine path compressions *done in postorder* and satisfying the rising roots condition requires linear time; i.e., given an initially postordered tree, each node in postorder is the subject of one left-spine path compression and is then deleted. Loebl and Nešetřil [17]–[19] prove linearity in a more general case, that of a sequence of left-spine path compressions satisfying the rising roots condition; they refer to this as a *local postorder*. These results derive from an open problem of Hart and Sharir [12]: What is the complexity of a sequence of path

compressions done in postorder without the rising roots condition? Both the Lucas and the Loebl–Nešetřil problems are special cases of this problem.

We now introduce a more general order of path compressions than that of either Lucas or Loebl and Nešetřil. Let $T^{(v)}$ be the subtree rooted at v just before the first path compression that includes v as a nonroot node and let $T_x^{(v)}$ be the subtree of $T^{(v)}$ rooted at x for some child x of v in $T^{(v)}$.

DEFINITION 4.6 (deque order). *An order-preserving sequence of path compressions is a* deque-ordered sequence *if and only if, for any v with two children x and y in $T^{(v)}$ such that $x \lessdot y$, the following is true: if the first path compression C' that orders v and y results in $y \lessdot v$, then all path compressions from nodes in $T_x^{(v)}$ precede C'; otherwise, if the first path compression C'' that orders v and x results in $v \lessdot x$, then all path compressions from nodes in $T_y^{(v)}$ precede C''.*

It is important to note that Definition 4.6 refers to nodes in a tree, e.g., $T_x^{(v)}$ with respect to $T^{(v)}$. That is, an actual compression may take place from a node that, at the time of the compression, is no longer a descendant of x (or v).

In the next section we prove our main result: a deque-ordered sequence of spine-only path compressions that satisfies the rising roots condition takes at most linear time.

5. Linearity of deque-ordered spine-only compression. We unfortunately find it necessary to divide the following definitions into two separate cases to handle differently what happens to the left and right of various nodes. Let $p_i(v)$ be the parent of v in T_i. For some path compression sequence $(C_1, \ldots, C_m)$ on T_0 and a node v, we call C_i a *left compression with respect to* v if

(1) C_i starts from a node x that is a descendant of v in T_0;

(2) C_i is a left-spine compression.

We define a *right compression with respect to* v symmetrically by replacing (2) with

(2′) C_i is a right-spine compression.

Note that x might not be a descendant of v at the time the path compression occurs. Let $\text{high}_L(v)$ be the shallowest node w that is a proper ancestor of v in T_0 and the root of a left compression with respect to v; if there is no such path compression, then $\text{high}_L(v) = \emptyset$. Symmetrically, let $\text{high}_R(v)$ be the shallowest node w that is a proper ancestor of v in T_0 and the root of a right compression with respect to v; $\text{high}_R(v) = \emptyset$ if no such compression exists. Again, it is critical to realize that the compression might not be from a descendant of v at the time.

Letting $\text{level}_L(\emptyset) = \text{level}_R(\emptyset) = -1$, we now define

$$\text{level}_L(v) = \text{level}_L(\text{high}_L(v)) + 1,$$
$$\text{last}_L(v) = \min\left\{\left\{i \mid v \text{ is not a proper descendant of } \text{high}_L(v) \text{ in } T_i\right\} \cup \{m+1\}\right\},$$
$$\text{level}_R(v) = \text{level}_R(\text{high}_R(v)) + 1,$$
$$\text{last}_R(v) = \min\left\{\left\{i \mid v \text{ is not a proper descendant of } \text{high}_R(v) \text{ in } T_i\right\} \cup \{m+1\}\right\}.$$

To make last_L and last_R well defined, we say that v is never a descendant of $\emptyset$. The definitions of level_L and level_R are static; they are defined purely with respect to the initial tree and the sequence of path compressions. A *left-level-defining descendant* of v is a descendant (if one exists) x of v in T_0 such that a left compression with respect to v from x roots at $\text{high}_L(v)$; we call the actual compression a *left-level-defining compression* of v. Symmetrically define a *right-level-defining descendant* and *right-level-defining compression* of v.

We now prove a set of technical lemmas that localize the compressions affecting a vertex to one or two left levels and one or two right levels adjacent to those of the vertex. Again we

let $p(v)$ ($p_i(v)$) be the parent of v (in T_i) (for v not the root) and $c_x(v)$ be the child of v that is an ancestor of x if x is a proper descendant of v.

LEMMA 5.1. *In a path compression sequence* $(C_1, \ldots, C_m)$ *on* T_0, *for any node* v,

1. *for any* $0 \leq i < \text{last}_L(v)$, $\text{level}_L(v) \geq \text{level}_L(p_i(v))$;
2. *for any* $0 \leq i < \text{last}_R(v)$, $\text{level}_R(v) \geq \text{level}_R(p_i(v))$.

Proof. We prove part (1) of the lemma; the proof for (2) is symmetric. In T_0, consider any root-to-leaf path partitioned by nodes of level_L zero. We use induction on the length of each partitioned subpath. The base case is the head r of the path; $\text{level}_L(r) = 0$. Note that $\text{level}_L(r) = 0 \Leftrightarrow \text{high}_L(r) = \emptyset \Leftrightarrow \text{last}_L(r) = 0$; the claim for the base case is thus vacuous.

For a node v such that $\text{level}_L(v) > 0$ and r is the nearest ancestor of v such that $\text{level}_L(r) = 0$, let x be a left-level-defining descendant of v. Any left compression with respect to v from x is also a left compression with respect to $p_0(v)$ from x. Thus, if $\text{high}_L(v) = r$ then $\text{high}_L(y) = r$ for every y such that y is an ancestor of v and a proper descendant of r in T_0. (No left compression with respect to r may root at a proper ancestor of r, since $\text{level}_L(r) = 0$.) By the definition of level_L, $\text{level}_L(v) \geq \text{level}_L(p_0(v))$. On the other hand, if $\text{high}_L(v) \neq r$ then $\text{high}_L(v)$ is a descendant of $\text{high}_L(p_0(v))$ in T_0. Both $\text{high}_L(v)$ and $\text{high}_L(p_0(v))$ are descendants of r in T_0. By the definition of level_L and the induction hypothesis, $\text{level}_L(v) \geq \text{level}_L(p_0(v))$.

For a node v in T_i such that $0 < i < \text{last}_L(v)$, by definition v is a proper descendant of $\text{high}_L(v)$ in T_i. By the above proof of monotonicity, it is still the case that $\text{level}_L(v) \geq \text{level}_L(p_i(v))$.

Leaf deletions trivially preserve part (1) of the lemma. □

LEMMA 5.2. *In a path compression sequence* $(C_1, \ldots, C_m)$ *on* T_0, *for any node* v *not the root of* T_0,

1. $\text{level}_L(p(v))$ *changes at most once during compressions* C_1 *through* $C_{\text{last}_L(v)-1}$;
2. $\text{level}_R(p(v))$ *changes at most once during compressions* C_1 *through* $C_{\text{last}_R(v)-1}$.

Proof. Again we prove part (1) of the lemma; the proof for (2) is symmetric. Let $\text{level}_L(v) = l$. Via the path compression sequence, v becomes a child of higher and higher nodes on the initial path from v to the root of T_0. By Lemma 5.1, $\text{level}_L(p(v))$ never increases during compressions C_1 through $C_{\text{last}_L(v)-1}$. By definition the highest node that becomes $p(v)$ during these compressions is of level_L not less than $l - 1$. Assuming $\text{last}_L(v) > 0$, Lemma 5.1 states that $\text{level}_L(p_0(v)) \leq \text{level}_L(v)$, and therefore $\text{level}_L(p(v))$ can change only once, from l to $l - 1$, during compressions C_1 through $C_{\text{last}_L(v)-1}$. □

Note that Lemmas 5.1 and 5.2 are completely general; they apply to any sequence of path compressions. We have not yet used the rising roots condition, spine-only compression, or deque order. The following lemma is critical to the counting argument used in the proof of Theorem 5.4, and it is here that we rely upon these conditions. We use the notation $T^{(v)}$ and $T_x^{(v)}$ as in Definition 4.6.

LEMMA 5.3. *In a deque-ordered sequence* $(C_1, \ldots, C_m)$ *of spine-only path compressions satisfying the rising roots condition, any node* v *can be in at most one compression of each of the following two types:*

1. (a) *The compression is a left-spine compression from a proper descendant* u *of* v;
   (b) $\text{level}_L(c_u(v)) = \text{level}_L(v) = \text{level}_L(p(v))$ *before the compression; and*
   (c) $p(v)$ *changes during the compression, but* $\text{level}_L(p(v))$ *does not.*
2. (a) *The compression is a right-spine compression from a proper descendant* u *of* v;
   (b) $\text{level}_R(c_u(v)) = \text{level}_R(v) = \text{level}_R(p(v))$ *before the compression; and*
   (c) $p(v)$ *changes during the compression, but* $\text{level}_R(p(v))$ *does not.*

Proof. Again we prove part (1) of the lemma; the proof for (2) is symmetric. Let $l = \text{level}_L(v)$. Consider the children of v in $T^{(v)}$; by the rising roots condition v can acquire

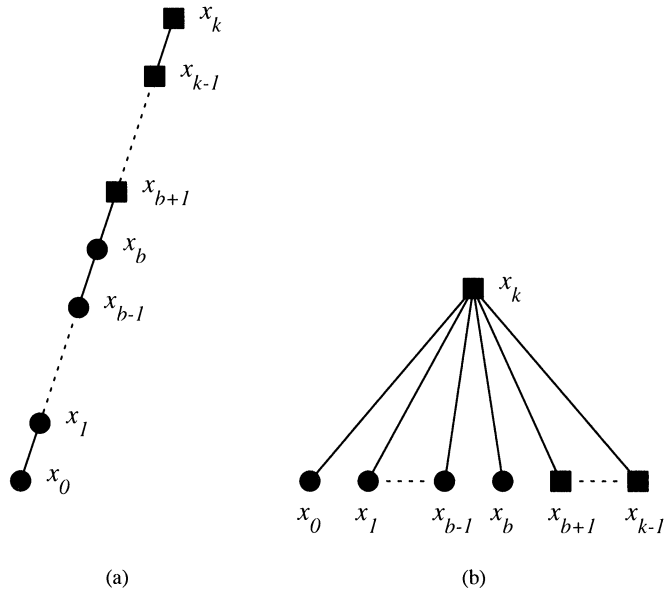


FIG. 5.1. (a) *A left spine (incident subtrees not shown); circles are nodes of* $\text{level}_L\ l$, *and squares are nodes of* $\text{level}_L\ l-1$. (b) *After compressing the path of* (a).

no more children. Furthermore, each compression as specified in the lemma removes a child from v. Now, if v has more than one child of $\text{level}_L\ l$ just before the first such compression, consider any two such children x and y such that $x \lessdot y$. Let x' be a left-level-defining descendant of x. Note that x' is a node in $T^{(v)}$; by order preservation, $c_{x'}(v) \lessdot y$ in $T^{(v)}$.

Consider the first compression C_i that orders y and v. If C_i makes $y \lessdot v$ (i.e., it is a left-spine compression), then by Definition 4.6 all the compressions from nodes in $T^{(v)}_{c_{x'}(v)}$ must precede C_i. In particular, the left-level-defining compression from x' rooting at a level_L $l-1$ node must occur before C_i. By the rising roots condition, C_i must therefore root at a node of level_L no greater than $l-1$; thus, if $\text{level}_L(p(v)) = \text{level}_L(v) = l$ just before C_i, then $\text{level}_L(p(v))$ changes (to $l-1$) during C_i. From that point until compression $C_{\text{last}_L(v)}$, $\text{level}_L(v) \neq \text{level}_L(p(v))$, completing the proof. Note that $i < \text{last}_L(v)$. □

We can now prove our main result.

THEOREM 5.4. *Any deque-ordered sequence* $(C_1, \ldots, C_m)$ *of* m *spine-only path compressions (and intermixed leaf deletions) satisfying the rising roots condition on an initial tree of* n *nodes incurs cost at most* $3m + 4n$.

Proof. Consider any left-spine path compression $C = (x_0, \ldots, x_k)$. The definition of level_L together with Lemma 5.1 shows that there are some b and l such that $\text{level}_L(x_i) = l$ for $0 \le i \le b$ and $\text{level}_L(x_i) = l-1$ for $b < i \le k$. If all the nodes on the path compression are of the same level_L, we say that $b = -1$. See Fig. 5.1(a).

Whereas before the compression, $p(x_i) = x_{i+1}$ for $0 \le i < k$, after the compression $p(x_i) = x_k$ for $0 \le i < k$. Each node x_i for $0 \le i < b$ has the level_L of its parent change, which by Lemma 5.2 can happen only once per node; we therefore charge each of these pointer changes to the nodes themselves. Furthermore, for all nodes x_i for $b+1 < i < k-1$, x_i is involved in the one type-(1) compression allowed by Lemma 5.3. Again, the related charges are incurred against the nodes themselves. Symmetric charges can be made against the nodes for a right-spine compression. The remaining charges, those for nodes x_b (if it exists), x_{b+1},

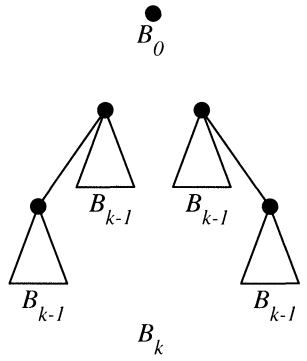


FIG. 6.1. *A B_0-tree and the two ways to make a B_k-tree.*

and x_{k-1}, are incurred against the compression itself. See Fig. 5.1(b). Since leaf deletions take unit time, the bound follows. □

It is not hard to see that the data structure of §3 uses a deque-ordered sequence of spine-only path compressions satisfying the rising roots condition. Thus, the proof of Theorem 5.4 completes the proof of Theorem 3.1. We again mention that the initial tree T_0 need not be balanced.

6. Further results. In this section we prove that general order-preserving path compression, even with the rising roots condition, can require superlinear time. Thus, adding further restrictions, such as deque order and spine-only path compression, is critical for achieving the above results. We also show how to achieve $O(\log^* n)$ worst-case time per heap-ordered deque operation when the number of deques is unbounded.

6.1. A lower bound for order-preserving path compression. In this section, we give an example demonstrating $2n$ order-preserving path compressions and leaf deletions satisfying the rising roots condition on an initial tree of size $2n$ with total cost $2n + n \log n$. Thus, at least one of the additional restrictions of spine-only path compression and deque order is crucial to the linearity of the path compression sequence used by the heap-ordered deque data structure.

To begin, we define the following trees inductively: a B_0-tree is a singleton node; a B_k-tree, $k > 0$, is formed by linking two B_{k-1}-trees as in Fig. 6.1. These trees resemble the binomial trees of Vuillemin [26]; since we allow linking in two directions, however, there are 2^k possible B_k-trees. It is easy to prove by induction that if T is a B_k-tree, then (1) $|T| = 2^k$, and (2) the depth of T is k.

Let L_k be a *collection* containing one B_i-tree for $0 \leq i \leq k$ laid out in the plane in some left-to-right order. We define a *deque order* on L_k inductively: L_0 is always *laid out in deque order*; L_k is *laid out in deque order* if and only if (1) the B_k-tree is the leftmost (or rightmost) tree, and (2) the rest of the trees form a collection L_{k-1} laid out in deque order to the right (or left) of the B_k-tree. See Fig. 6.2. We use the following lemma in our lower bound construction.

LEMMA 6.1. *A collection L_k laid out in deque order with the root of each B_i-tree in the collection connected to a common parent x forms a B_{k+1}-tree.*

Proof. The proof is by induction. The base case for $k = 0$ is trivial. For $k > 0$, consider a collection L_k laid out in deque order. Assume without loss of generality that the B_k-tree T_l is leftmost. By the induction hypothesis, the collection L_{k-1} laid out in deque order to the right of T_l together with the node x forms a B_k-tree T_r. The root of T_l, however, is the leftmost child of x, the root of T_r, and so by definition this structure forms a B_{k+1}-tree. See Fig. 6.2. □

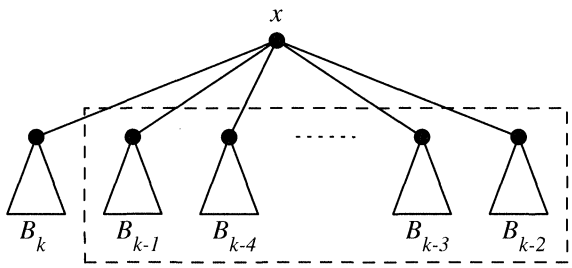


FIG. 6.2. *A collection L_k laid out in deque order. The roots of the B_i-trees are children of a common node x. The trees inside the box together with x form a B_k-tree as described in Lemma* 6.1.

We *augment* a B_k-tree by making its root the child of a new root node. Call the augmented tree an A_k-tree; see Fig. 6.3(a). Let r be the root of an A_k-tree. By definition, the leftmost (and rightmost) child of r is the root r_{k-1} of a B_{k-1}-tree. Either the leftmost or rightmost child of r_{k-1} is the root r_{k-2} of a B_{k-2}-tree. Continuing in this way, we derive a path $P = (s_0, r_0, r_1, \ldots, r_{k-1}, r)$ in the A_k-tree, where r_i is the root of a B_i-tree (and s_0 is the root of a B_0-tree). Note that there are two B_0-trees in the unwound construction. Furthermore, each node in P other than r is an extremal child of its parent. See Fig. 6.3(b). Thus, P can be compressed in an order-preserving way.

LEMMA 6.2. *Compressing P in an order-preserving way yields a collection L_{k-1} of trees laid out in deque order and an extra B_0-tree, all connected to the common node r.*

Proof. After the compression, the leftmost (or rightmost) child of r is r_{k-1} (descending from which is the B_{k-1}-tree). By induction, the B_i-trees descending from nodes r_i, $0 \leq i < k-1$, form a collection L_{k-2} laid out in deque order to the right (or left) of the B_{k-1}-tree; their roots, along with s_0, all become children of r. See Fig. 6.3(c). □

By compressing P, we can change the A_k-tree into a B_k-tree and an extra node. Deleting the extra node and repeating the procedure yields our result.

THEOREM 6.3. *A sequence of n order-preserving path compressions and leaf deletions satisfying the rising roots condition on an initial tree of n nodes can incur $\Theta(n \log n)$ cost.*

Proof. First, we derive the lower bound. Form the initial tree T by augmenting a B_k-tree n times, where $n = 2^k$. This produces a B_k-tree attached to a path $(x_1, \ldots, x_n)$ of n nodes as depicted in Fig. 6.4. Form the path P as used in Lemma 6.2 from the B_k-tree and node x_1. By Lemma 6.2, compressing path P yields a collection L_{k-1} of trees laid out in deque order and an extra B_0-tree, all of the roots of which are children of x_1. Deleting the extra B_0-tree, which is just a singleton node, yields a B_k-tree, the root of which is a child of node x_2; Lemma 6.1 shows this. We can perform this procedure n times, each time performing a path compression of length $k+1$ and a leaf deletion. The path compressions are order preserving and satisfy the rising roots condition. Initially, $|T| = 2n$ and $k = \log_2 n$, thus proving the lower bound.

Since the path compressions satisfy the rising roots condition, Lemma 4.2 shows that the path compression sequence corresponds to an instance of disjoint set union with path compression. Tarjan and van Leeuwen [25] prove that such a sequence incurs $O(n \log n)$ cost. □

6.2. A worst-case upper bound per heap-ordered deque operation. Theorems 3.1 and 5.4 prove that our heap-ordered deque data structure requires $O(1)$ amortized time per operation. The worst-case time per operation on the data structure can easily be linear, however, or at best $O(\log n)$ (for a heap-ordered deque of n elements) if we link heap-ordered deques by size.

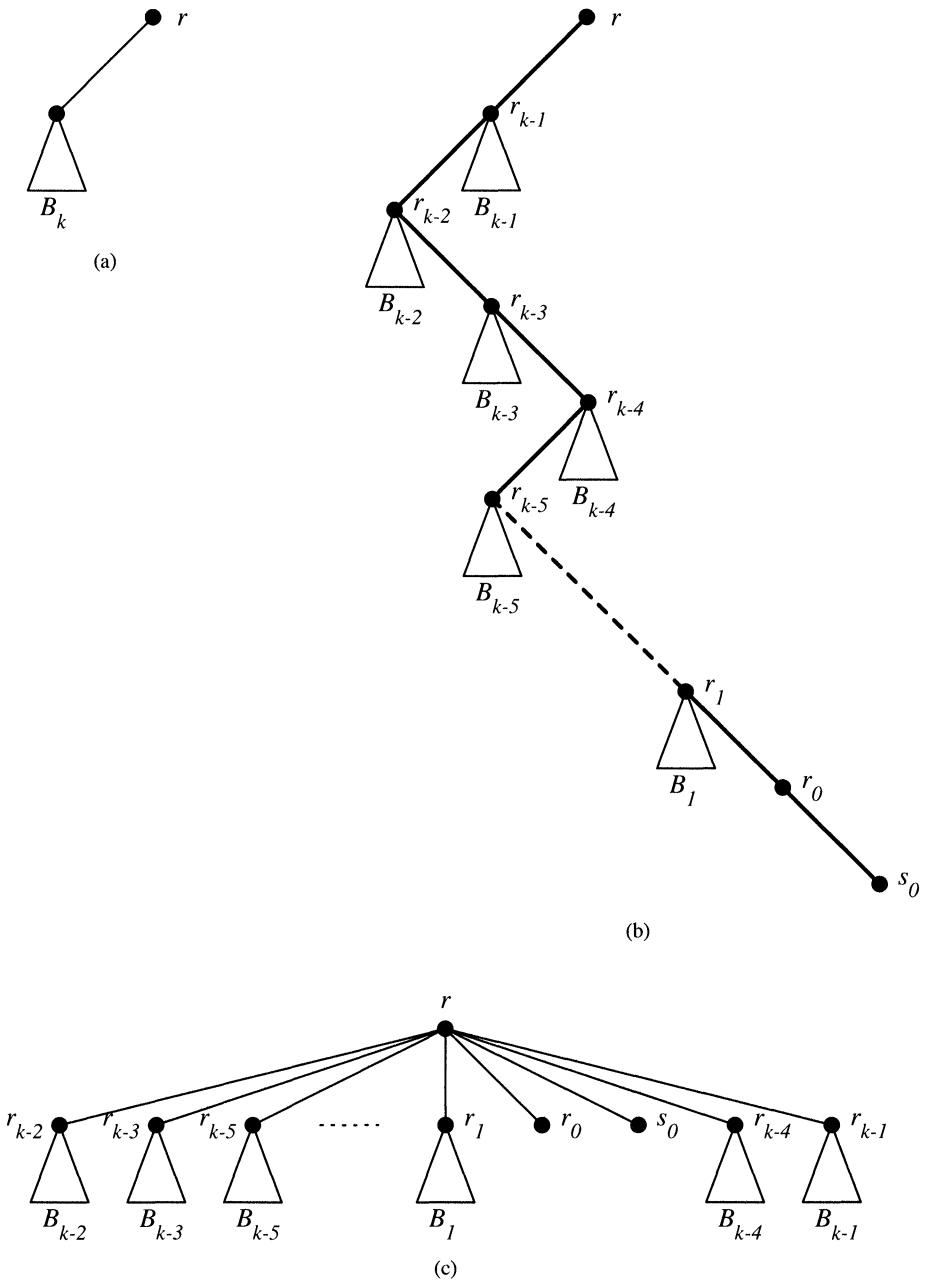


FIG. 6.3. (a) *An A_k-tree.* (b) *An A_k-tree unwound to show path P in bold.* (c) *After compressing path P.*

Buchsbaum and Tarjan [5] recently used two types of data-structural bootstrapping to design confluently persistent deques. One of the key ideas of that work is the *spine decomposition*, which breaks a tree into paths and represents those paths by catenable deques. If the leaves of the tree represent elements of a deque and the tree maintains balance, then the deques representing the paths of the tree are strictly smaller than the deque represented by the tree itself. (In fact, they are logarithmic in the size of the original deque.) They call this type

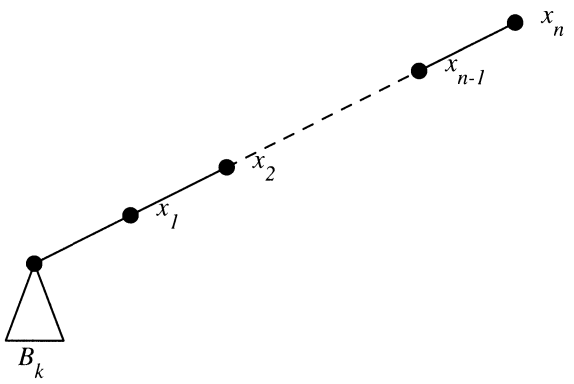


FIG. 6.4. *A B_k-tree augmented n times.*

of bootstrapping (i.e., homogeneously decomposing a data structure into smaller instances) *structural decomposition*. Via structural abstraction, they represent the tree paths (and associated lists of paths) by persistent version numbers. The result is confluently persistent deques of n elements that require only $O(\log^* n)$ amortized time per operation. (They also give a worst-case time solution.)

We remark here that the same decomposition is possible for trees representing heap-ordered deques. Instead of representing the tree paths and associated lists by version numbers, we represent them by their minimum elements. Thus, we can obtain the same $O(\log^* n)$ time bound per operation for catenable heap-ordered deques. The underlying persistence machinery produces the amortization in the confluently persistent deque bound. For the heap-ordered deque application, the time bound is worst case, not amortized. As discussed below, Kosaraju [15] has recently discovered an $O(1)$ worst-case time solution that uses additional ideas.

7. Conclusion and open problems. We have described how to implement catenable heap-ordered double-ended queues in constant amortized time per operation (worst case if the number of queues is fixed); we have also given an $O(\log^* n)$ worst-case time per operation solution for an arbitrary number of queues. The important pieces of our work are the use of the bootstrapping technique of Driscoll, Sleator, and Tarjan [8] in designing the data structure and the analysis of deque-ordered spine-only path compression in proving its efficiency. In particular, our path compression result generalizes the work of Loebl and Nešetřil [17]-[19] and Lucas [20] and provides what we believe is the first application for such results.

In a preliminary version of this paper [4], we asked if there was an implementation of catenable heap-ordered deques that achieves constant worst-case time per operation (for an arbitrary number of deques). Kosaraju [15] subsequently designed such a data structure. His solution begins with an alternative structure, based on finger search trees, that requires $O(1)$ amortized time per operation. He combines this with structural abstraction in a clever two-tiered data structure that maintains an intricate balance invariant. None of the known solutions is immediately extensible to the confluently persistent deque problem of Buchsbaum and Tarjan [5], however: our solution has amortized complexity, and Kosaraju's worst-case solution uses catenable double-ended queues deep in the data structure. Extending any of these methods to give an $O(1)$-time implementation of confluently persistent deques remains an open problem. (See note added in proof.)

Data-structural bootstrapping holds promise as a general tool for designing data structures with some sort of *join* operation together with a property secondary to the original data

structure. Driscoll, Sleator, and Tarjan [8] use structural abstraction to effect persistence in catenable lists; we employ it to effect heap order in catenable deques. Buchsbaum and Tarjan [5] use structural abstraction and structural decomposition to produce confluently persistent deques. Work on formalizing these methods and finding further applications seems worthwhile.

The Hart-Sharir [12] open problem of postorder path compression without the rising roots condition remains tantalizingly open. We note that our lower bound construction in §6.1 does not involve any order on the compressions themselves. Furthermore, the construction produces an unbalanced initial tree. Thus, one can still formulate intermediate open problems incorporating different notions of order as well as the absence or presence of rising roots and tree balance that might be easier to solve than the Hart-Sharir problem.

Note added in proof. Since this article went to press, Haim, Kaplan, and Tarjan have devised an algorithm to implement confluently persistent output-restricted deques with constant worst-case time per operation, via a technique called *recursive slowdown*. This result will appear in the 1995 Symposium on the Theory of Computing. They have extended this result to the unrestricted case as well. Chris Okasaki has discovered a completely different solution to the problem, with a constant time bound per operation that is amortized rather than worst case. So far his result is limited to the output-restricted case.

Acknowledgments. Jeff Westbrook initially brought the problem of catenable heap-ordered deques to our attention. Rao Kosaraju provided patient insight into the problem. Milena Mihail and Peter Winkler rendered thoughtful criticism of an earlier draft of §§2 and 3. Haim Kaplan corrected some minor problems with the formalism in §5. Greg Frederickson pointed us to Booth and Westbrook [3].

REFERENCES

[1] P. K. Agarwal, A. Aggarwal, B. Aronov, S. R. Kosaraju, B. Schieber, and S. Suri, *Computing external farthest neighbors for a simple polygon*, Discrete Appl. Math., 31 (1991), pp. 97-111.

[2] C. R. Aragon and R. G. Seidel, *Randomized search trees*, in Proc. 30th IEEE Symposium on Foundations of Computer Science, Research Triangle Park, NC, 1989, pp. 540-545.

[3] H. Booth and J. R. Westbrook, *A linear algorithm for analysis of minimum spanning and shortest-path trees of planar graphs*, Algorithmica, 11 (1994), pp. 341-352.

[4] A. L. Buchsbaum, R. Sundar, and R. E. Tarjan, *Data structural bootstrapping, linear path compression, and catenable heap ordered double ended queues*, in Proc. 33rd IEEE Symposium on Foundations of Computer Science, Pittsburgh, PA, 1992, pp. 40-49.

[5] A. L. Buchsbaum and R. E. Tarjan, *Confluently persistent deques via data structural bootstrapping*, in Proc. 4th ACM-SIAM Symposium on Discrete Algorithms, Austin, TX, 1993, pp. 155-164; J. Algorithms, to appear.

[6] R. Cole and A. Siegel, *River routing every which way, but loose*, in Proc. 25th IEEE Symposium on Foundations of Computer Science, Singer Island, FL, 1984, pp. 65-73.

[7] G. Diehr and B. Faaland, *Optimal pagination of B-trees with variable-length items*, Comm. Assoc. Comput. Mach., 27 (1984), pp. 241-247.

[8] J. R. Driscoll, D. D. K. Sleator, and R. E. Tarjan, *Fully persistent lists with catenation*, in Proc. 2nd ACM-SIAM Symposium on Discrete Algorithms, 1991, pp. 89-99; J. Assoc. Comput. Mach., 41 (1994), pp. 943-959.

[9] J. Françon, G. Viennot, and J. Vuillemin, *Description and analysis of an efficient priority queue representation*, in Proc. 19th IEEE Symposium on Foundations of Computer Science, Ann Arbor, MI, 1978, pp. 1-7.

[10] G. N. Frederickson, *Planar graph decomposition and all pairs shortest paths*, J. Assoc. Comput. Mach., 38 (1991), pp. 162-204.

[11] H. Gajewska and R. E. Tarjan, *Deques with heap order*, Inform. Process. Lett., 22 (1986), pp. 197-200.

[12] S. Hart and M. Sharir, *Nonlinearity of Davenport–Schinzel sequences and of generalized path compression schemes*, Combinatorica, 6 (1986), pp. 151-177.

[13] D. S. HIRSCHBERG AND L. L. LARMORE, *New applications of failure functions*, J. Assoc. Comput. Mach., 34 (1987), pp. 616-625.

[14] S. R. KOSARAJU, *Real-time simulation of concatenable double-ended queues by double-ended queues*, in Proc. 11th ACM Symposium on Theory of Computing, Atlanta, GA, 1979, pp. 346-351.

[15] ———, *An optimal RAM implementation of catenable min double-ended queues*, in Proc. 5th ACM-SIAM Symposium on Discrete Algorithms, Arlington, VA, 1994, pp. 195-203.

[16] L. L. LARMORE AND D. S. HIRSCHBERG, *Efficient optimal pagination of scrolls*, Comm. Assoc. Comput. Mach., 28 (1985), pp. 854-856.

[17] M. LOEBL AND J. NEŠETŘIL, *Linearity and unprovability of set union problem strategies*, in Proc. 20th ACM Symposium on Theory of Computing, Chicago, IL, 1988, pp. 360-366.

[18] M. LOEBL AND J. NEŠETŘIL, *Postorder hierarchy for path compressions and set union*, in Proc. 5th International Meeting of Young Computer Scientists, Lecture Notes in Computer Science 381, Springer-Verlag, Berlin, New York, Heidelberg, 1988, pp. 146-151.

[19] ———, *Linearity and unprovability of set union problem strategies* I: *Linearity of on line postorder*, Tech. report 89-04, Department of Computer Science, University of Chicago, 1989.

[20] J. M. LUCAS, *Postorder disjoint set union is linear*, SIAM J. Comput., 19 (1990), pp. 868-882.

[21] E. M. MCCREIGHT, *Pagination of B*-trees with variable-length records*, Comm. Assoc. Comput. Mach., 20 (1977), pp. 670-674.

[22] R. SUNDAR, *Worst-case data structures for the priority queue with attrition*, Inform. Process. Lett., 31 (1989), pp. 69-75.

[23] R. E. TARJAN, *Data Structures and Network Algorithms*, in CBMS-NSF Regional Conference Series in Applied Mathematics, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[24] ———, *Amortized computational complexity*, SIAM J. Algebraic Discrete Meth., 6 (1985), pp. 306-318.

[25] R. E. TARJAN AND J. VAN LEEUWEN, *Worst-case analysis of set union algorithms*, J. Assoc. Comput. Mach., 31 (1984), pp. 245-281.

[26] J. VUILLEMIN, *A data structure for manipulating priority queues*, Comm. Assoc. Comput. Mach., 21 (1978), pp. 309-315.

[27] ———, *A unifying look at data structures*, Comm. Assoc. Comput. Mach., 23 (1980), pp. 229-239.

SIAM J. COMPUT.
Vol. 24, No. 5, pp. 1207-1234, December 1995

006

ADAPTIVE PATTERN MATCHING*

R. C. SEKAR†, R. RAMESH‡, AND I. V. RAMAKRISHNAN§

Abstract. Pattern matching is an important operation used in many applications such as functional programming, rewriting, and rule-based expert systems. By preprocessing the patterns into a deterministic finite state automaton, we can rapidly select the matching pattern(s) in a single scan of the relevant portions of the input term. This automaton is typically based on left-to-right traversal of the patterns. By *adapting* the traversal order to suit the set of input patterns, it is possible to considerably reduce the space and matching time requirements of the automaton. The design of such adaptive automata is the focus of this paper. We first formalize the notion of an adaptive traversal. We then present several strategies for synthesizing adaptive traversal orders aimed at reducing space and matching time complexity. In the worst case, however, the space requirements can be exponential in the size of the patterns. We show this by establishing an exponential lower bound on space that is *independent* of the traversal order used. We then discuss an orthogonal approach to space minimization based on *direct* construction of optimal directed acyclic graph (dag) automata. Finally, our work stresses the impact of typing in pattern matching. In particular, we show that several important problems (e.g., lazy pattern matching in ML) are computationally difficult in the presence of type disciplines, whereas they can be solved efficiently in the untyped setting.

Key words. pattern matching, indexing, discrimination nets, functional programming, algorithms and complexity

AMS subject classifications. 68N15, 68Q25, 68Q40, 68Q42

1. Introduction.

Pattern matching is a fundamental operation in a number of important applications such as functional and equational programming [11], [21], term rewriting, theorem proving [7], and rule-based systems [6]. In most of these applications, patterns are partially ordered by assigning priorities. For instance, in languages such as ML [10] and Haskell [13], a pattern occurring earlier in the text has a higher priority over those patterns following it. In HOPE [3] and in many rule-based systems, more specific patterns have higher priority over less specific ones [6], [16], [20]. In theorem-proving applications, priorities are used to encode efficient heuristics. Applications that do not impose priorities can be handled as a special case by assigning equal priorities to all patterns.

The typical approach to fast pattern matching is to preprocess the patterns into a deterministic finite state automaton that can rapidly select the patterns that match the input term. The main advantage of such a matching automaton is that all pattern matches can be identified in a *single* scan (i.e., no backtracking) of portions of input terms relevant for matching purposes and is done in time that is *independent* of the number of patterns. Existing automaton-based approaches (as well as the approach we present in this paper) do not handle nonlinear patterns (i.e., patterns with multiple occurrences of the same variable). This is because many applications use only linear patterns; and even for those applications that permit nonlinear patterns, it has been observed that most failures are associated with symbol mismatches [4], which can be detected without taking nonlinearity into account.

Figure 1 shows a matching automaton constructed on the basis of a left-to-right traversal of patterns. Each state of the automaton corresponds to the prefix of the input term seen in reaching that state and is annotated with the set of patterns that can possibly match. For

*Received by the editors March 26, 1993; accepted for publication (in revised form) June 15, 1994.

†Bellcore, Rm 2A-274, 445 South Street, Morristown, New Jersey 07962 (sekar@thumper.bellcore.com). This author's research was completed at SUNY, Stony Brook and supported by National Science Foundation grants CCR-8805734 and 9102159 and NYS S&T grant RDG 90173.

‡Department of Computer Science, University of Texas at Dallas, Richardson, Texas 75083 (ramesh@utdallas.edu). This author's research was supported by National Science Foundation grant CCR-9110055.

§Department of Computer Science, State University of New York at Stony Brook, Stony Brook, New York 11794 (ram@cs.sunysb.edu). This author's research was supported by National Science Foundation grants CCR-9102159, 9404921, CDA-9303181, and INT-9314412.

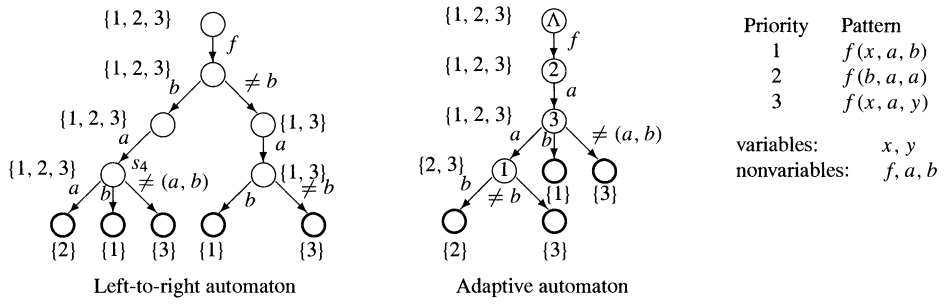


FIG. 1. *Automata for* $f(x, a, b)$, $f(b, a, a)$, $f(x, a, y)$.

instance, state s_4 corresponds to having inspected the prefix $f(b, a, x)$, where x denotes the subterm that has not yet been examined. This state is annotated with the pattern set $\{1, 2, 3\}$ since we cannot rule out a match for any of the three patterns on the basis of the prefix $f(b, a, x)$.

Pattern matching automata have been studied extensively for over a decade. Christian [4] obtained dramatic speedups in a Knuth-Bendix completion system by using an automaton for unprioritized patterns based on left-to-right traversal. Graf [9] describes a similar automaton for unprioritized patterns. In functional programming, Augustsson [1], [2] and Wadler [26] describe pattern matching techniques that are based on left-to-right traversal, but which also deal with priorities.

The methods of Augustsson and Wadler are economical in terms of space usage, but may reexamine symbols in the input term. In the worst case, these methods can degenerate to the naive method of testing each pattern separately against the input term. In contrast, the methods of Christian [4], Graf [9], and Schnoebelen [24] avoid reexamining symbols, but achieve this at the cost of increasing the space requirements. In fact, Graf and Schnoebelen show that the upper bounds on the size of their automata are exponential.

One way to improve both space *and* matching time is to engineer a traversal order to suit the set of patterns or the application domain. We refer to such traversals as *adaptive traversals* and to automata based on such traversals as *adaptive automata*. Because traversal orders are no longer fixed a priori, an adaptive automaton must specify the traversal order. For instance, in the adaptive automaton shown in Fig. 1, each state is labeled with the next argument position to be inspected from that state. Adaptive traversal has the following advantages over a *fixed-order* of traversal such as left-to-right:

1. The resulting automaton is usually smaller, e.g., the adaptive automaton in Fig. 1 has 8 states compared to 11 in the left-to-right automaton. The reduction factor can even become *exponential.*

2. Pattern matching requires lesser time with adaptive traversals than fixed-order traversals. For example, the left-to-right automaton needs to inspect four symbols to announce a match for pattern 1. The adaptive automaton inspects only a subset of these positions (three of them) to announce the match. Examining unnecessary symbols is especially undesirable in the context of lazy functional languages since it runs counter to the goals of lazy evaluation.

Unlike automata based on fixed-order traversals, relatively little is known about designing automata based on adaptive traversals. There are a number of interesting questions that arise in such a design. For instance, given a set of patterns, how do we choose a traversal order to realize the advantages mentioned above? Is it possible to select a traversal order that improves space and matching time over automata based on fixed-traversal orders? What are the lower and upper bounds on space and matching time complexities of such automata? Finally, in the

context of functional programming, using an arbitrary order of traversal for pattern matching is inappropriate because it affects the termination properties of the program. Given this constraint, is it possible to internally change the traversal order so that it does not affect the termination properties and at the same time realize the advantages of adaptive traversal? We answer all these questions in this paper. In addition, we also solve some of the problems that have remained open even in the context of left-to-right traversals. Such problems include

1. tight bounds on space and matching time complexity,
2. impact of type discipline on the computational complexity of many problems that arise in construction of matching automata,
3. direct construction of optimal automata that shares equivalent states.

1.1. Summary of results. In §2, we first formalize the concept of adaptive traversal and its special case, fixed traversal orders. We then present a generic algorithm for building an adaptive automaton that is parameterized with respect to the traversal order. Our generic algorithm is used as a basis for the results presented in later sections. As mentioned before, our algorithms assume that the patterns are linear, i.e., no variable in any pattern occurs more than once. The automaton can still be used for nonlinear patterns, but on reaching a final state we would have to check for consistency of the substitutions for the different occurrences of the same variable.

In §3, we present several techniques to synthesize traversal orders that can improve space and matching time:

1. We develop the important concept of a *representative set* which forms the basis of several optimization techniques aimed at avoiding inspection of unnecessary symbols.
2. Next we present several powerful strategies for synthesizing traversal orders. Through an intricate example, we show that they all can sometimes increase both space and matching time.
3. We then show that a strategy that inspects *index* positions whenever possible does not suffer from the above drawback. Specifically we show that inspecting index positions can only improve the space and matching time of the automata. Huet and Lévy [14] established the importance of index in the design of an optimal automaton (in the sense of not seeing any unnecessary symbols) for strongly sequential patterns. Our results extend the applicability of indices even for patterns that are not strongly sequential.
4. Finally, we study the synthesis of traversal orders that are appropriate for lazy functional languages. In such languages, pattern matching is closely coupled with evaluation so that the traversal order used for matching can affect the termination properties of the program. Therefore, the programmer must be made aware of the traversal order T used for matching. The question then is whether we can synthesize a traversal order that has the same (or better) termination properties as T. In §3.5, we show how to synthesize such a traversal $S(T)$ that enables the programmer to assume T; an implementation can benefit from significant improvements in space and time using the adaptive traversal $S(T)$.

In §4, we discuss the computational aspects of the strategies presented in §3. Our work clearly brings forth the impact of typing in pattern matching. We have shown that several important problems in the context of pattern matching are unlikely to have efficient algorithms in typed systems; we give polynomial-time algorithms for them in untyped systems. In particular, we do the following:

1. We present a quadratic-time algorithm for computing representative sets in untyped systems; we show that computing these sets is *NP*-complete for typed systems.
2. In §4.2 we focus on the important problem of index computation in prioritized systems. Laville [18] and Puel and Suárez [23] have extended Huet and Lévy's [14] index computation algorithm to deal with priorities. However, all these algorithms require exponen-

tial time in the worst case. In contrast, we show that indexes can be computed in polynomial time in the case of untyped systems.

3. We also show that index computation in typed systems is *co-NP*-complete.

In §5, we examine the space and matching time complexity of the adaptive automata. We show that the space requirements of the automata can be quite large by establishing the first known tight exponential lower bounds on size.

In §6, we describe an orthogonal approach to space minimization based on directed acyclic graph (dag) representation. By tightly characterizing the equivalent states of the adaptive automata, we directly build their optimal dag representation. Since the unoptimized automaton can be exponentially larger than the optimized one, a direct construction can use polynomial space and time; a method that uses finite state automaton (FSA) minimization techniques may require exponential time and space. As mentioned in [9], this important problem of direct construction had remained open even for left-to-right traversals.

Finally, we conclude in §7 with a discussion of the various strategies presented in the paper and how they can be combined to build efficient adaptive automata.

2. Preliminaries. In this section we develop the notations and concepts that will be used in the rest of this paper. We also present a generic algorithm for construction of an adaptive automaton. This generic algorithm forms the basis for the results presented in later sections.

We assume familiarity with the basic concept of a *term*. The symbols in a term are drawn from a nonempty *ranked alphabet* Σ and a countable set of variables $\mathcal{X}$. (The term *arity* is sometimes used in the literature in place of *rank*.) We will use a, b, c, d, and f to denote nonvariable symbols and x, y, and z (with or without subscripts and primes) to denote variables. We use '_' to denote unnamed variables. Each occurrence of '_' denotes a distinct variable. Let *root*(t) denote the symbol appearing at the root of a term t. In order to refer to subterms of a term, we develop the following concept of a *position*.[1]

DEFINITION 2.1 (position). *A position is either*

1. *the empty string* Λ *that reaches the root of the term, or*
2. $p.i$, *where* p *is a position and* i *an integer, which reaches the* i*th argument of the root of the subterm reached by* p.

If p is a position then the subterm of t reached by p is denoted by t/p. We use $t[p \leftarrow s]$ to denote the term obtained from t by replacing t/p by s. The set of variable positions in a term t is known as the *fringe* of t. We illustrate these concepts using the term $t = f(a(x), b(a(y), z))$. Here, $t/\Lambda = t, t/2 = b(a(y), z), t/2.1 = a(y)$, and $t/2.2 = z$. The term $t[2 \leftarrow c] = f(a(x), c)$ is obtained by replacing the second argument of f by (the term) c. The fringe of t is $\{1.1, 2.1.1, 2.2\}$.

A *substitution* is a mapping from variables to terms. Given a substitution β, an *instance* $t\beta$ of t is obtained by substituting $\beta(x)$ for every variable x in t. If t is an instance of u then we say $u \leq t$ and call u a *prefix* of t. The inverse of $\leq$ relation is denoted by $\geq$. For the term $t = f(a(x), b(y, z))$ and the substitution β that maps x to $b(x', x'')$, y to c, and z to itself, $t\beta = f(a(b(x', x'')), b(c, z))$.

We say that two terms t and s unify, denoted $t \uparrow s$, iff they possess a common instance. $t \sqcup s$ denotes the least such instance in the ordering given by $\leq$. For example, the terms $t = f(a(x), y)$ and $s = f(x', b(y', z'))$ possess common instances $f(a(x''), b(y'', z''))$, $f(a(c), b(y'', z''))$, $f(a(x''), b(a(y''), z''))$, etc. The least common instance $t \sqcup s$ is the term $f(a(x''), b(y'', z''))$.

[1] The terms *occurrence* and *path* are sometimes used in the literature to denote the same concept.

In the rest of the paper, we use $\mathcal{L}$ to denote the given set of patterns. (A *pattern* is simply a term.) All patterns are assumed to be *linear*, i.e., no variable in them appears more than once. Given such a set and the priority relationship among the patterns, we formalize the notion of pattern matching as follows:

DEFINITION 2.2 (pattern match). *A pattern $l \in \mathcal{L}$ matches u iff $l \leq u$, and no $l' \in \mathcal{L}$ with priority greater than that of l unifies with u. If there is any $l \in \mathcal{L}$ that matches u then we say that there is a* pattern match *for u.*

Note that this definition differs from the traditional notion of matching in that it takes priorities into account. The traditional notion only requires that the term u be an instance of the pattern l. Here, we also require that u not unify with a pattern of higher priority. The intuition behind this requirement is that u actually denotes a prefix of the term t that is being inspected to identify a match.[2] If it unifies with a pattern of higher priority, then we may identify a match for this higher-priority pattern when we inspect some of the symbols below this prefix. Since we cannot rule out a match for any higher-priority pattern without inspecting the symbols below u, we cannot declare a match for l. Also note that, by definition of a pattern match, when we do identify a match for u, we can announce a match for t itself.

To illustrate the concept of a pattern match, consider once again the set of patterns in Fig. 1. Only the first pattern matches the term $f(a, a, b)$. Note that no pattern matches the term $f(x, a, a)$ although it is an instance of the third pattern. This is because $f(x, a, a)$ unifies with the second pattern, which has a higher priority than the third one.

Given a term u, we define its *match set* $\mathcal{L}_u$ as the set of patterns that unify with u. Observe that by definition of a pattern match, we can restrict ourselves to $\mathcal{L}_u$ if we are looking for a match for any term with a prefix u. Furthermore, to identify a match, we can restrict ourselves to inspection of only those fringe positions $\mathcal{F}_u$ wherein at least one of the patterns in $\mathcal{L}_u$ has a nonvariable. In this context, some of the standard traversal orders such as depth-first and breadth-first traversals will also be modified to skip fringe positions that are not in $\mathcal{F}_u$. For illustration of these concepts, consider $\mathcal{L} = \{f(x, a, b), f(x, a, a), f(x, a, y)\}$ and $u = f(x', y', a)$. In this case, $\mathcal{L}_u = \{f(x, a, a), f(x, a, y)\}$. Note that the only variable position in u where every pattern in $\mathcal{L}_u$ has a variable is position 1, and so $\mathcal{F}_u = \{2\}$.

In our search for efficient pattern-matching algorithms, we will need the following concept of an index position, which is a fringe position that *must* be inspected by any matching algorithm to announce a match for a term. Formally, we have the following definition.

DEFINITION 2.3 (index). *A fringe position p for a prefix u is said to be an* index *with respect to a pattern l iff for every term $t \geq u$ such that l matches t, t/p is a nonvariable. It is said to be an index of u with respect to a set of patterns $\mathcal{L}$ iff it is an index with respect to every $l \in \mathcal{L}$.*

The notion of an index is closely related to the concept of sequentiality [15], [14]. One simple example of an index is a fringe position of u where every pattern in $\mathcal{L}_u$ has a nonvariable. Observe that such a position must be inspected to determine if the given term is an instance of any of the patterns in $\mathcal{L}_u$.

We now classify patterns depending on how they affect the complexity of various problems related to pattern matching. We first classify based on whether more than one pattern can match any given term.

[2] The variables in u denote unexamined positions in t. We deliberately blur the distinction between a variable and uninspected portions of a term. This is because, in linear terms, a variable simply stands for any arbitrary term or unknown term. This correctly reflects our intuition that we have no knowledge of the term structure below a variable in the prefix u.

DEFINITION 2.4 (ambiguous and unambiguous patterns). *A set of patterns $\mathcal{L}$ is said to be* ambiguous *whenever there is a term t such that more than one pattern matches it. Otherwise $\mathcal{L}$ is* unambiguous.

Our next classification is based on type discipline. In typed systems, the set of allowable input terms is constrained by a type discipline. From a pattern-matching perspective, this constraint requires that arguments to a function symbol f be drawn from a specific set of terms, say, $\{T, F\}$. In this case, terms $f(T, F)$, $f(F, T)$, $f(T, T)$, and $f(F, F)$ are permissible (or well typed) whereas $f(2, T)$ is not. In untyped systems, there are no restrictions on the arguments of functions. For instance, the terms $f(2, F)$ and $f(c, d)$ are valid input terms. (A more rigorous treatment of types can be readily found in the literature, but is not developed here since the results presented in this paper can be established based only on the simple distinction given here.) The time and space complexity of many problems discussed in this paper will vary widely based on whether the system is typed and also on whether it is ambiguous.

Next we define the notion of a traversal order. We call a traversal order *top down* if it visits a node only after visiting its ancestors. Since we are interested only in identifying matches at the root of the input term, we deal with only top-down traversals and do not concern ourselves with bottom-up traversals such as those used in [12]. Any traversal order has associated with it a characteristic function. This function specifies the position to be inspected after having inspected a prefix u. For example, a left-to-right traversal has the following characteristic function. Having visited a prefix u, this function chooses the leftmost position in $\mathcal{F}_u$ as the next position to visit. Note that if the prefix visited is simply a variable (i.e., no symbols have been inspected yet), then all traversal orders will specify the root position as the next one to inspect. We distinguish fixed and adaptive traversals using the following definition.

DEFINITION 2.5 (adaptive and fixed traversals). *An* adaptive *traversal is a top-down traversal wherein the position $p \in \mathcal{F}_u$ next visited is a function of the prefix u and the set $\mathcal{L}_u$. In a fixed traversal order, p is simply a function of $\mathcal{F}_u$.*

By choosing the next position as a function of $\mathcal{L}_u$, an adaptive traversal can *adapt* itself to the given set of patterns. In contrast, a fixed-order traversal always makes a fixed choice from the positions in $\mathcal{F}_u$. For instance, a left-to-right traversal would always pick the leftmost position in $\mathcal{F}_u$. A breadth-first traversal would pick the leftmost position among positions of least length (i.e., the length of the integer string representing a position) from those in $\mathcal{F}_u$. Having defined the notion of adaptive traversals, we now proceed to present a generic algorithm for building adaptive automata.

2.1. Generic algorithm to build adaptive automata. Figure 2 shows our algorithm *Build* for constructing an adaptive automaton. A state v of the automaton remembers the prefix u inspected in reaching v from the start state. Suppose that p is the next position inspected from v. Then there are transitions from v on each distinct symbol c that appears at p for any $l \in \mathcal{L}_u$. There will also be a transition from v on $\neq$ which will be taken on inspecting a symbol different from those on the other edges leaving v. The symbol $\neq$ appearing at a position p denotes the inspection of a symbol in the input that does not occur at p in any pattern in $\mathcal{L}_u$. This implies that if a prefix u has a $\neq$ at a position p then every pattern that could potentially match an instance of u must have a variable at or above p.

There is an important distinction between *Build* and previously known algorithms (such as that of Christian [4] and Huet and Lévy [14]). *Build* is nondeterministic in that it does not specify a selection function to specify the next position to visit. This nondeterminism enables us to reason about automaton construction *without any reference* to the traversal order used.

Procedure *Build* is recursive, and the automaton is constructed by invoking *Build*(s_0, x) where s_0 is the start state of the automaton. *Build* takes two parameters: v, a state of the automaton, and u, the prefix examined in reaching v. The invocation *Build*(v, u) constructs

Procedure $Build(v, u)$
1. Let $\mathcal{M}$ denote patterns that match u.
2. If $\mathcal{M} \neq \phi$ and $\forall\, l \in \mathcal{L}_u \;\; \exists l' \in \mathcal{M}$ such that $priority(l') \geq priority(l)$ then
3. $\quad match[v] := \mathcal{M}$. /* State v announces a match for patterns in $\mathcal{M}$ */
4. else
5. $\quad p = select(u)$ /* *select* is a function to choose the next position to inspect */
6. $\quad pos[v] = p$ /* Next position to inspect is recorded in the *pos* field */
7. $\quad$ for each symbol for which $\exists l \in \mathcal{L}_u$ with $root(l/p) = c$ (for a nonvariable c) do
8. $\quad\quad$ create a new node v_c and an edge from v to v_c labeled c
9. $\quad\quad Build(v_c, u[p \leftarrow c(y_1, \ldots, y_{rank(c)})])$
10. $\quad$ If $\exists l \in \mathcal{L}_u$ with a variable at p or at an ancestor of p then
11. $\quad\quad$ create a new node $v_{\neq}$ and an edge from v to $v_{\neq}$ labeled $\neq$
12. $\quad\quad Build(v_{\neq}, u[p \leftarrow \neq])$

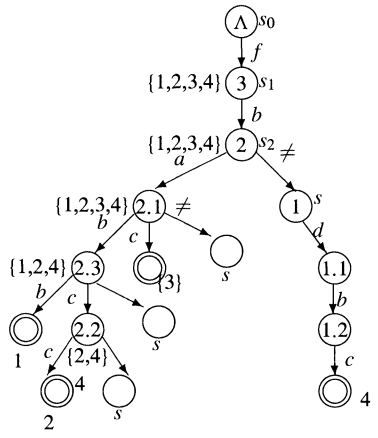
FIG. 2. *Algorithm for constructing adaptive automaton.*

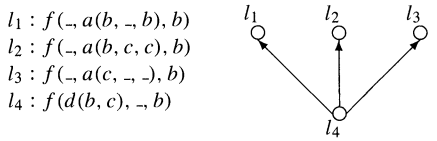
the subautomaton rooted at v. In line 2, the termination conditions are checked. By definition of pattern match, we need to rule out possible matches with higher-priority patterns before declaring a match for a lower-priority pattern. Since the match set $\mathcal{L}_u$ contains all patterns that could possibly match the prefix u, we simply need to check that each pattern in the match set is either already in $\mathcal{M}$ or has a lower priority than some pattern in $\mathcal{M}$.

If the termination conditions are not satisfied then the automaton construction is continued in lines 5 through 12. At line 5, the next position p to be inspected is selected and this information is recorded in the current state in line 6. Lines 7, 8, and 9 create transitions based on each symbol that could appear at p for any pattern in $\mathcal{L}_u$. In line 9, *Build* is recursively invoked with the prefix extended to include the symbols seen on the transitions created in line 8. If there is a pattern in $\mathcal{L}_u$ with a variable at or above p, then a transition on $\neq$ is created at line 11 and *Build* is recursively invoked at line 12. The recursive calls initiated at lines 9 and 12 together will complete the construction of the subautomaton rooted at state v. Steps 10-12 will be skipped in case of typed systems if we have created transitions at Step 9 corresponding to every symbol that can appear at the position p.

To illustrate the algorithm, consider the set of patterns shown in Fig. 3. Pattern l_4 has a lower priority than any of the other patterns. There is no relationship among the priorities of l_1, l_2, and l_3. The automaton construction begins with the invocation $Build(s_0, x)$. Observe that none of the patterns match the prefix x. Since the only fringe position in prefix x is Λ, $pos[s_0]$ is set as Λ. The state s_1 is created at line 8 and the edge between s_0 and s_1 is labeled by symbol f. Following this, a recursive call $Build(s_1, f(x, y, z))$ is made at line 9. At line 2 of this recursive call, we once again find that no pattern matches $f(x, y, z)$. Now assume that select at line 5 returns 3. Note that all patterns have the same nonvariable symbol at position 3 and so there is only one iteration of the loop at line 7. This results in the invocation of $Build(s_2, f(x, y, b))$. Continuing this process, we obtain the automaton shown in Fig. 3. The automaton has 25 states and each unlabeled state marked by s represents a subtree identical to that rooted at the state labeled by 1.

Procedure *Build* as given is inefficient in that it computes the sets $\mathcal{L}_u$ and $\mathcal{M}$ in each recursive call. It also manipulates positions, which are *strings* of integers. The first source of inefficiency can be overcome easily by computing the match sets incrementally. To overcome the second source of inefficiency, positions can be encoded as integers in the range $[1, S]$, where S is the sum of the sizes of all patterns. A similar but indirect technique is used in the context of compilation of pattern matching for functional programming languages [2], [22], [19].





1. States are labeled by the next position to be inspected upon reaching that state.
2. All of the states marked with an s actually denote a subautomaton that is identical to the automaton rooted at the state labeled with 1.

FIG. 3. *Adaptive automaton constructed by Build.*

3. Synthesizing traversal orders. Observe that *Build* does not specify the selection function. In this section we present several techniques that can be used to construct the selection functions. These techniques can be either used independently or combined as appropriate.

3.1. Measuring size and matching time. The primary objective of a selection function is to reduce the automaton size and/or the matching time. Therefore it is important to know how we measure these quantities. A natural measure of the size of an automaton is the number of states in it. Unfortunately, minimization of total number of states is *NP*-complete, even for the simple case of patterns with no variables [5]. This makes it impossible to develop efficient algorithms that build an automaton of smallest size, unless $P = NP$. Even so, we would still like to show that certain algorithms are always better than others for reducing the size. The usual way to do this is to choose an alternative measure of size that is closely related to the original size measure yet does not have the drawback of *NP*-completeness of its minimization. A natural choice in this case is the breadth of the automaton, which is related to the total number of states by at most a linear factor. (In practice, however, the factor is typically much smaller: note that if every node in the automaton has at least two children, then the breadth is at least half as much as the number of states.)

As for matching time, it is easy to define the time to match a given term using a given automaton: it is simply the length of the path in the automaton from the root to the final state that accepts the given term. However, what we would like is a time measure that does not refer to input terms. We could associate an average matching time with an automaton, but this would require information that is not easily obtained: the relative frequencies with which each of the paths in the automaton are taken.[3] Therefore, instead of defining a time measure that totally orders the automata for a specific distribution of input terms, we use the following measure that partially orders them *independently* of the distribution. Let $MT(s, A)$ denote the length of the path in (automaton) A from the start state to the accepting state of the *ground* term s. (A ground term is a term that contains no variables.) If s is not accepted by A then $MT(s, A)$ is undefined.

DEFINITION 3.1. *Let A and A' be two matching automata for $\mathcal{L}$, and t be any term. We say that $A \preceq_t A'$, meaning that A is more efficient than A' for matching instances of t, iff for every ground instance s of t, $MT(s, A) \leq MT(s, A')$.*

[3] It is possible to assume that all terms over Σ are equally likely and derive a matching time on this basis, but such assumptions are seldom justified or useful in practice.

$A \preceq A'$ is a shorthand for $A \preceq_x A'$, where x is a variable. Note that $A \preceq A'$ means that A is more efficient than A' for matching *every* ground term.

When we need to get an idea of the work involved in matching various classes of patterns, it is useful to associate a quantitative measure of matching time with automata. We do this by first identifying the best possible automaton for the given set of patterns (as given by the partial order $\preceq$) and computing the average root-to-leaf path length of this automaton. When multiple minimal automata with different average root-to-leaf path lengths are possible, we will refrain from giving a quantitative measure.

3.2. Representative sets. In this section, we introduce the important concept of a *representative set*. This notion makes use of the priorities among the patterns to eliminate those patterns from the match set for which no matches can be found. This happens for a prefix u whenever a match for a lower priority pattern implies a match for a higher priority pattern. For instance, consider the patterns in Fig. 3 and the prefix $u = f(\_, a(b, \_, b), \_)$. Although $\mathcal{L}_u = \{l_1, l_4\}$, observe that a match for l_4 can be declared only if the third argument of f is b. In such a case we declare a match for the higher-priority pattern l_1. Inspecting any position only on behalf of a pattern such as l_4 is wasteful, e.g., inspection of position 1 for u is useless since it is irrelevant for declaring a match for l_1. We can avoid inspecting such positions by considering the representative set instead of a match set for a prefix u. A representative set is defined formally as follows.

DEFINITION 3.2. *A representative set $\overline{\mathcal{L}_u}$ of a prefix u with respect to a set of patterns $\mathcal{L}$ is a minimal subset $\mathcal{S}$ such that the following condition holds for every l in $\mathcal{L}$:*

$$(1) \quad \forall t \geq u \ \ (l \text{ matches } t) \Rightarrow \exists l' \in \mathcal{S} \ \ [(l' \text{ matches } t) \wedge (\text{priority}(l') \geq \text{priority}(l))].$$

Using the definition of pattern match and through simplification, we can derive the following simpler condition equivalent to (1). We use the notation $P(l)$ to denote the priority of pattern l.

$$(2) \quad \forall t \geq (l \sqcup u)\ \exists l' \in \mathcal{S} \ \ [(P(l') > P(l)) \wedge (l' \uparrow t)] \vee [(P(l') = P(l)) \wedge (l' \leq t)].$$

This condition captures our intuition about l that for any instance of u, either l does not match the instance (first part of the disjunction) or the match can be "covered" by another pattern in $\mathcal{S}$ (second part). We refer to the property given by this condition as a *cover* property, and any set $\mathcal{S}$ satisfying the property as a *cover* for $\mathcal{L}$. A representative set is simply a minimal cover. We make the following observation about the transitivity of the cover property.

Observation 1. If $\mathcal{S}_1$ is a cover for $\mathcal{L}$ and $\mathcal{S}_2$ is a cover for $\mathcal{S}_1$ then $\mathcal{S}_2$ is a cover for $\mathcal{L}$.

Note that if the set $\mathcal{L}$ contains multiple patterns with equal priority, then there may be a choice as to which of these patterns are retained in a representative set. Thus the definition does not always yield a unique representative set. Hence future references to $\overline{\mathcal{L}_u}$ or *representative set* refer to any one set that satisfies the above definition.

Laville's notion of accessible patterns [18] is similar to our notion of representative sets, but is not a minimal set. Our contributions here are that the minimality enables us to develop more efficient algorithms and that we provide an algorithm for computing this set. The definition of accessible patterns in [18] does not yield such an algorithm and so Laville uses the notion of compatible patterns (which corresponds to our match set $\mathcal{L}_u$) in place of accessible patterns.[4]

[4]We can also use $\mathcal{L}_u$ in place of $\overline{\mathcal{L}_u}$ in all the optimizations mentioned in this paper, but doing so may make the optimizations less effective. For instance, our algorithm for directly building optimal dag automata in §6 will fail to identify some equivalent states if $\mathcal{L}_u$ is used in place of $\overline{\mathcal{L}_u}$.

In the following section, we show how to design selection strategies using representative sets. The algorithmic aspects of computing the representative sets will be discussed in §4.

3.3. Greedy strategies. In this section we present many strategies for implementing the function *select* at a node v of the automaton. All these strategies select the next position based on *local information* such as the prefix and the representative set associated with v or its children. Let p denote the next position to be selected. z

1. Select a p such that the number of distinct nonvariables at p, taken over all patterns in $\overline{\mathcal{L}_u}$, is *minimized.* This strategy attempts to minimize the size by local minimization of breadth of the automata. It does not attempt to reduce matching time.

2. Select a p such that the number of distinct nonvariables at p taken over all patterns in $\overline{\mathcal{L}_u}$ is *maximized.* The rationale here is that by maximizing the breadth, a greater degree of discrimination is achieved. If we can quickly distinguish among the patterns, then the (potentially) exponential blow-up can be contained. Furthermore, once we distinguish one pattern from the others, we no longer inspect unnecessary symbols and thus matching time can also be improved.

3. Select a p such that the number of patterns having nonvariables at p is maximized. The motivation for this strategy is that only patterns with variables at p are duplicated in the representative sets of the descendents of the current state v of the automaton. By minimizing the number of duplicated patterns, we can contain the blow-up. Furthermore, this choice minimizes the probability of inspecting an unnecessary position: it is a necessary position for the most number of patterns.

4. Let $\overline{\mathcal{L}_1}, \ldots, \overline{\mathcal{L}_r}$ be the representative sets of the children of v. Select a p such that $\Sigma_{i=1}^{r} |\overline{\mathcal{L}_i}|$ is minimized. Note that the main reason for exponential blow-up is that many patterns get duplicated among the representative sets of the children of v. This strategy locally minimizes such duplication (since $\Sigma_{i=1}^{r} |\overline{\mathcal{L}_i}|$ is given by the size of $\overline{\mathcal{L}_u}$ plus the number of patterns that are duplicated among the representative sets of the children states.) For improving time, this strategy again locally minimizes the number of patterns for which an unnecessary symbol is examined at each of the children of v.

All of the above greedy strategies suffer from the following drawback.

THEOREM 3.3. *For each of the above strategies there exist pattern sets for which automata of smaller size can be obtained by making a choice different from that given by the strategy.*

Proof. It is quite straightforward to give an example for strategy 1 and we omit it here. For strategies 2, 3, and 4 consider the following set of patterns with equal priorities:

$$f(a, a, \_, \_) \quad f(b, \_, a, \_) \quad f(c, \_, \_, a),$$
$$f(\_, b, b, b) \quad f(\_, c, c, c) \quad f(\_, d, d, d).$$

After inspecting the root, all these strategies will choose one of positions 2, 3, or 4. It can be shown (by enumerating all possible matching automata for these patterns) that the smallest breadth and number of states obtainable by this choice are 20 and 49, respectively. These figures can be reduced to 15 and 45 respectively by choosing position 1. □

The construction of this example is quite intricate. The key idea is to make each of the pattern sets $\{1, 4, 5, 6\}$, $\{2, 4, 5, 6\}$, and $\{3, 4, 5, 6\}$ strongly sequential,[5] whereas any set containing two of the first three patterns and one of the last three is not strongly sequential. Such a choice ensures that if a traversal order first inspects position 1 after the root position, then every subautomata below this state will match a set of strongly sequential patterns. Since optimal automata can be constructed for such patterns, these subautomata can be of a small

[5] In strongly sequential systems, any prefix u with $|\mathcal{L}_u| \geq 1$ must have an index.

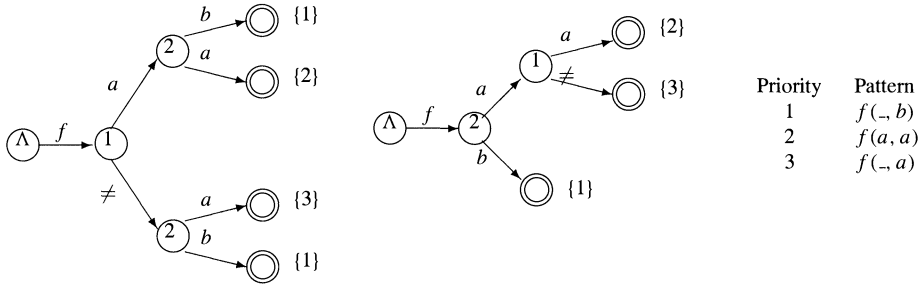


FIG. 4. *Illustration of size and matching time reduction due to interchange step.*

size. In contrast, any other choice of position to inspect will result in match sets that are not strongly sequential, and hence lead to subautomata that are larger.

The contrived nature of the example shows that although it is possible for these strategies to fail, such failures may be rare. Even when they fail, as in the above example, they still appear to be significantly better than fixed-order traversals. For instance, right-to-left traversal constructs an automaton with breadth 30 and number of states 60. Left-to-right traversal reduces these figures to 24 and 58, which is still more than that obtained by using strategies 2, 3, and 4.

Conceptually, the problems outlined above, with regard to choice of next inspected position, arise in the context of minimizing matching time as well. However, since the definition of $\preceq$ does not permit us to compare arbitrary automata, such a result cannot be established without making assumptions about the distribution of input terms.[6]

3.4. Selecting indices. We now propose another important local strategy that does not suffer from the drawbacks of the greedy strategies discussed in the previous section. The key idea is to inspect the index positions in u whenever they exist. We show that this strategy yields automata of smaller (or same) size and superior (or same) matching time than that obtainable by any other choice. The importance of an index was known only in the context of strongly sequential systems. Our result demonstrates its applicability to patterns that are not strongly sequential.

Consider the two automata shown in Fig. 4. The second automaton is obtained from the first by interchanging the order in which positions 1 and 2 are visited. Observe that by inspecting the index position 2 before the nonindex position 1, the second automaton improves the space requirements. It also requires less time to identify a match since each path in the second automaton examines only a subset of the positions examined in the corresponding path(s) in the first automaton. The following theorem formalizes the interchange operation and outlines its benefits.

THEOREM 3.4. *Let v be a state in an automaton B, the prefix corresponding to v be u, and $pos[v] = p$. If q is an index of u such that $pos[w] = q$ for every child w of v then we can obtain an automaton B' from B by interchanging the order of inspection of p and q in such a way that $|B'| \leq |B|$ and $B' \preceq B$.*

Proof. First we describe the construction of B' from B. Let A be the subautomaton of B that is rooted at v and u be the prefix that has been inspected in reaching v. The construction of B' takes place in two steps.

[6] One must not, however, conclude our choice of $\preceq$ is inappropriate for comparing matching times. As mentioned before, it is essentially the only way to compare matching time when we have no knowledge of the distribution of input terms.

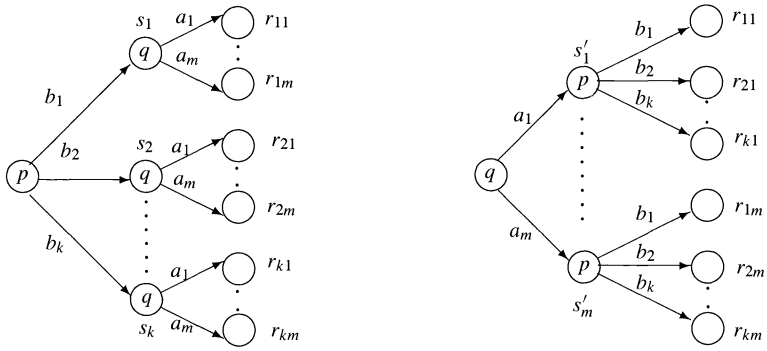


FIG. 5. *Automaton before and after an interchange step.*

Step 1: Interchange the order of inspection of positions p and q, without changing any other part of the automaton, as shown in Fig. 5. In this figure $a_1, \ldots, a_m$ are all the symbols that appear at q in any pattern in $\overline{\mathcal{L}_u}$. Similarly, $b_1, \ldots, b_k$ are all the symbols that appear at p for any pattern in $\overline{\mathcal{L}_u}$. Note b_k is $\neq$ if some pattern in $\overline{\mathcal{L}_u}$ has a variable at or above q. It is also possible for a_m to be $\neq$. Also note that some of the states r_{ij} may not be present because no pattern in $\mathcal{L}_u$ has a b_i at p and a_j at q. Such r_{ij}'s denote empty subautomata. Note that the prefix inspected at r_{ij} is the same in the automaton before and after the interchange. Hence the structure of the subautomata rooted at an r_{ij} are also identical before and after the interchange. Consequently, it is possible to perform the above interchange in the order of inspection of p and q.

Step 2: Replace each s'_i that satisfies the following condition by r_{i1}:

$$\forall l \in \overline{\mathcal{L}_{prefix(s'_i)}} \quad l/p \text{ is a variable,}$$

where the notation $prefix(s'_i)$ is used to denote the prefix inspected on reaching the state s'_i of the automaton. States such as s'_i that satisfy the above condition can exist whenever p is *not* an index. When this condition is satisfied, all the subautomata rooted at $r_{1i}, r_{2i}, \ldots, r_{mi}$ that are below s'_i will be identical and the position p need not be inspected at all. Therefore, in the second step, we replace such s'_i by r_{1i}.

This completes the construction of B'. We now show that $|B'| \leq |B|$. This is easy to see. Step 1 does not change the breadth of the automaton (which is our measure of size) and Step 2 can only reduce the breadth.

To show that $B' \preceq B$, we proceed as follows. The construction above (Steps 1 and 2) induces a mapping $\mathcal{I}$ from the final states in B to final states in B' (which are a subset of final states in B). For a final state v_1 in B, its image $v'_1 = \mathcal{I}(v_1)$ in B' is defined as follows:

Case 1: v_1 is a descendent of an r_{ij} such that the state s'_i was eliminated (by replacing it with r_{i1}) in Step 2 of the construction. By remarks made in Step 2, r_{ij} are identical for $1 \leq j \leq m$. This identity defines a natural correspondence between final states in an r_{ij} and final states in r_{i1}. We take $\mathcal{I}(v_1)$ in this case to be the corresponding state within r_{i1}.

Case 2: Otherwise (i.e., not Case 1), v_1 appears unchanged in B' and so we take $\mathcal{I}(v_1) = v_1$.

Now consider any term s for which a match is announced by B and let v_1 be the final state reached. It is easy to see that the state $\mathcal{I}(v_1)$ will be reached when B' is used for matching s. Moreover, the positions examined on the path to $\mathcal{I}(v_1)$ is a subset of those inspected in reaching v_1. By definition of $\preceq$, this implies that $B' \preceq B$. □

Observe that the above theorem can be used only if all children of v inspect the same index position q. However, even if some children of v inspect position q' other than q, it is still possible to globally rearrange the automaton to inspect q at v and thus achieve the benefits of inspecting the index position early. This is because q is an index position and hence must be inspected before declaring a match. Therefore it will appear in each path from v to any accepting state in the subautomaton rooted at v.

THEOREM 3.5. *Let v be a state in an automaton B, u be the prefix corresponding to v, and $pos[v] = p$. If q is an index of u then we can obtain another matching automaton B' from B by replacing its subautomaton A rooted at v by another (sub)automaton A' such that $|B'| \leq |B|$ and $B' \preceq B$.*

Proof. We construct B' by repeating the interchange operation. The proof that B' can be so obtained is by straightforward induction on the height of the subautomaton A. □

Although the theorem only asserts that size and matching time of A' is no larger than that of A, Fig. 4 shows that they can be strictly smaller for A'. We remark that by repeating the interchange steps as above, size can sometimes be reduced by as much as an exponential factor and time by $O(n)$, where n is the number of patterns.

We point out that the interchange operations mentioned above merely constitute a proof technique; they play no part at all in the actual construction of the automaton. In the actual construction, the same effect is obtained by simply modifying the selection function to inspect indices whenever possible.

3.5. Adaptive traversal orders for lazy functional languages. In lazy functional languages, evaluation (of input terms) is closely coupled with pattern matching. Specifically, a subterm of the input term is evaluated only when its root symbol needs to be inspected by the pattern matcher. If there are subterms whose evaluation does not terminate, then an evaluator that uses an algorithm that identifies matches without inspection of such subterms can terminate, whereas use of algorithms that do inspect such subterms will lead to nontermination. Since the set of positions inspected to identify a match is dependent on the traversal order used, the termination properties also depend upon the traversal order. In order to make sure that the program terminates on input terms of interest to the programmer, the programmer (sometimes) has to reason about the traversal order used. In particular, the programmer can code his/her program in such a way that (for terms of interest to him/her) the pattern matcher will inspect only those subterms whose evaluation will terminate. This implies that the programmer must be made aware of the traversal order used *even before the program is written*—thereby ruling out synthesis of *arbitrary* traversal orders at compile time. Given this constraint on preserving termination properties, a natural question is whether the traversal order can be "internally changed" by the compiler in a manner that is transparent to the programmer. Specifically, given a traversal order T that is assumed by the programmer, our goal is to synthesize a new traversal order $S(T)$ such that any evaluation algorithm that terminates with T will also do so with $S(T)$. We devise such a traversal order in this section.

3.5.1. Preliminaries. We first tighten our original definition of a traversal so as to capture the important aspect of determinacy in the order in which various positions are visited.

Condition 1 (*monotonicity*). Suppose that a traversal order T selects p as the next position to be visited for a prefix u. T is said to be monotonic iff for any prefix $u' \geq u$, the following condition holds: If u'/p is a variable and l/p is a nonvariable in some $l \in \overline{\mathcal{L}_{u'}}$ then the traversal once again selects p.

Monotonic traversals include most known traversal orders such as depth-first and breadth-first, as well as variations of these without left-to-right bias. An example of a traversal that is not monotonic is given by the following select function:

$$\text{select}(f(x, y, z)) = 1, \tag{3}$$

$$\text{select}(f(x, a, z)) = 3. \tag{4}$$

We also require that the selection function make its decision only based on portions of the prefix that are "relevant," as given below.

Condition 2. Let u_1 and u_2 be two prefixes with identical representative sets. Suppose that the prefixes differ only in subterms appearing at positions where every pattern in this representative set has a variable. Then the selection function must choose the same position to inspect for u_1 and u_2.

Clearly, the symbols appearing at such positions are irrelevant for determining a match—and consequently irrelevant for selecting which positions to inspect next. Therefore we require that the selection function choose the same position to inspect for u_1 and u_2 in such a case. Henceforth, we only consider selection functions that satisfy the above two conditions.

Given a monotonic traversal T, we define $S(T)$ as follows.

DEFINITION 3.6. *$S(T)$ is any traversal order characterized by the following selection function for any prefix u:*

1. *If u has indices then arbitrarily choose one of them.*
2. *Otherwise, choose the position that would be selected by T.*

3.5.2. Size and matching time improvement using $S(T)$. Let B_0 be a matching automaton that uses traversal order T. We will now show that any automaton B' using $S(T)$ can be constructed from B_0 through the interchange operations of Theorem 3.4 and 3.5. It then follows from Theorem 3.5 that $S(T)$ improves space and matching time requirements over an automaton using T. Also note that (as established in the proof of Theorem 3.4) after the interchange step, each path in the new automaton examines a subset of the positions examined in the corresponding path in the old automaton. Therefore, a lazy evaluation algorithm based on the new automaton will terminate in every case when the algorithm terminated with the old automaton.

For the proof that the interchange operations lead to an automaton using $S(T)$, we need to go back to the construction in Fig. 5. Recall that v is a state in B corresponding to a prefix u with $pos[v] = p$, and for every child w of v, $pos[w]$ is an index position q. Let B' be the automaton obtained by interchanging the order of inspection of p and q. To relate the traversal orders used in B and B', we define the following correspondence, mapping $\mathcal{C}$ from each state v' of B' that inspects a nonindex position to a set of states in B.

Case 0: $v' = v$. Since v inspects an index in B', $\mathcal{C}(v)$ need not be defined.

Case 1: either v' is not a descendent of v, or v' is a descendent of a state r_{ij} such that the state s'_i was not eliminated in Step 2. It is clear that in this case, the state appears unchanged in B and so $\mathcal{C}(v') = \{v'\}$.

Case 2: $v' = s'_i$ for some i such that s'_i was *not* eliminated in Step 2. Then $\mathcal{C}(v') = \{v\}$.

Case 3: v' is a descendent of an r_{i1} such that s'_i was eliminated by Step 2 of the construction. Since $r_{i1}, \ldots, r_{im}$ are identical in this case, there is a natural one-to-one correspondence between states in r_{ij} and those in r_{i1}. $\mathcal{C}(v')$ in this case will be the set of all the states in $r_{i1}, \ldots, r_{im}$ that correspond to v' in this manner.

We make the following observations about the mapping $\mathcal{C}$.

Observation 3.7. For every v' in B' that examines a nonindex position

$$\forall s \in \mathcal{C}(v') \quad pos[v'] = pos[s].$$

Although this observation shows that the positions inspected in corresponding states are identical, it does not imply anything about the traversal orders used, since no assertion is made

about the prefixes inspected at these states. To establish a relationship between the selection functions used in B and B', we show the following result.

LEMMA 3.8. *Let B' be an automaton obtained by performing one interchange step on B. Let v' be any state in B' that examines a nonindex position. Also let sel be the selection function used in B in any state that does not choose an index to inspect next, i.e., for any state s in B with $pos[s]$ not an index of $prefix(s)$, $sel(prefix(s)) = pos[s]$. Then*

$$pos[v'] = sel(prefix(v')).$$

Proof. The proof is by analysis of each of the cases defining C. In Case 1, the prefix inspected at v' is identical to that inspected at the (only) state s in $C(v')$. Thus $pos[v'] = pos[s] = sel(prefix(s)) = sel(prefix(v'))$. In Case 2, it is easy to see that for the only element v in $C(v')$, $prefix(v') = prefix(v)[q \leftarrow a_i(x_1, \ldots, x_{rank(a_i)})]$ for some symbol a_i. Also note that $prefix(v')/p$ is a variable since p has not been inspected in reaching v'. In addition, it is not possible for every pattern in $\overline{\mathcal{L}_{prefix(v')}}$ to have a variable at or above p - in such a case, the state v' would have been eliminated by Step 2 of our construction. These facts, together with monotonicity, imply that $p = sel(prefix(v')) = sel(prefix(v)) = pos[v] = pos[v']$. In Case 3, let s be any state in $C(v')$. It is easy to see (by Step 2 of construction) that $prefix(s)$ is identical to $prefix(v')$ in all positions except p. Moreover, by criteria for applying Step 2 of construction of B', every pattern $l \in \overline{\mathcal{L}_{prefix(v')}}$ has a variable at or above p. Therefore the symbol at p is irrelevant for pattern matching. By Condition 2 on selection function, $sel(prefix(v'))$ must be the same as $sel(prefix(s)) = pos[s] = pos[v']$. □

We now extend Lemma 3.8 so it holds for arbitrary number of interchange steps between B and B'.

LEMMA 3.9. *Let B' be an automaton obtained by performing zero or more interchange steps on B_0. Let v' be any state in B' that examines a nonindex position. Also let sel be the selection function used in B_0 in any state that does not choose an index to inspect next, i.e., for any state v_0 in B_0 with $pos[v_0]$ not an index of $prefix(v_0)$, $sel(prefix(v_0)) = pos[v_0]$. Then*

$$pos[v'] = sel(prefix(v')).$$

Proof. The proof is by simple induction on the number of interchange steps used to obtain B' from B_0. □

LEMMA 3.10. *Let T be a monotonic traversal order. Any automaton using $S(T)$ can be obtained by performing the interchange operations on an automaton that uses T.*

Proof. Let B_0 be an automaton that uses traversal order T. Use the construction outlined in Lemmas 3.8 and 3.9 repeatedly on the automaton B_0 to obtain another automaton B' that examines indices as early as possible. In this automaton B', if a state v' does not inspect an index position then it inspects the position specified by selection function used in B_0, i.e., a position given by the traversal order T (see proof of Lemma 3.9). By Definition 3.6, this means that B' uses $S(T)$, so we need only show that the above construction can be used to obtain an automaton for *any* traversal order that follows Definition 3.6. Note that $S(T)$ is uniquely determined by T, except for the order in which index positions are inspected. Clearly, all possible permutations of such positions can be obtained using the interchange operation. Thus, any automaton that follows Definition 3.6 can be obtained through interchange operations from B_0. □

THEOREM 3.11. *Let T be a monotonic traversal. Size and matching time can never become worse if $S(T)$ is used in place of T. Moreover, each path in the automaton using $S(T)$ examines a subset of the positions examined on the corresponding path in T.*

Proof. Since the interchange operations can only improve space and matching time, the first part of the theorem is immediate. For the second part, note that, by construction of the

interchange operation, the positions inspected on a root-to-leaf path in the automaton after the interchange are a subset of those positions inspected before the interchange. □

We remark that the subset property ensures that any evaluation algorithm that terminates with traversal order T will terminate if $S(T)$ is used in place of T.

4. Computational aspects. In order to implement the selection function described in the previous section we must develop an algorithm to compute the representative set. Similarly we must have a method to identify indices of a prefix to incorporate space and matching time optimizations. We discuss the algorithmic aspects of these problems in the following section.

4.1. Computing representative sets. We now present an efficient procedure for computing representative sets in untyped systems.

Procedure $computeRepSet(u, \mathcal{L}_u)$
1. $\mathcal{L}' := \mathcal{L}_u$
2. While $\exists l_1, l_2 \in \mathcal{L}'$ $[(l_1 \neq l_2) \wedge (l_1 \sqcup u \geq l_2) \wedge (priority(l_2) \geq priority(l_1))]$ do
3. delete l_1 from $\mathcal{L}'$

THEOREM 4.1. *Procedure computeRepSet computes the representative set of u in $O(nS)$ time for untyped systems where n is the number of patterns in $\mathcal{L}_u$ and S the sum of sizes of these patterns.*

Proof. The time complexity result can be readily established, so we focus only on correctness. We first establish that $\mathcal{L}'$ is a cover by inducting on the number of times the loop at lines 2-3 is executed. In the base case $\mathcal{L}' = \mathcal{L}_u$ is obviously a cover. For the induction step, let $\mathcal{L}_1$ and $\mathcal{L}_2$ denote the values of $\mathcal{L}'$ before and after the deletion of l. By induction hypothesis and transitivity of the cover property, we need only show that $\mathcal{L}_2$ is a cover for $\mathcal{L}_1$. Given the condition at line 2 of the algorithm, it is easy to see that the cover condition given in (2) holds with $l = l_1, l' = l_2, \mathcal{L} = \mathcal{L}_1$, and $\mathcal{S} = \mathcal{L}_2$.

We now prove by contradiction that the set $\mathcal{L}'$ returned by *computeRepSet* is a minimal cover. Assume that it is not, so there must be an $l \in \mathcal{L}'$ that can be deleted without affecting the cover property. This assumption means that for every $t \geq l \sqcup u$, the body of condition (2) holds with $l' \neq l$. Now consider the term t obtained by instantiating all variables in $l \sqcup u$ by $\neq$. By construction of t, if any pattern l' unifies with t then it must be that $t \geq l'$, and $l \sqcup u \geq l'$. Thus, the body of condition (2) implies that $\exists l' \in \mathcal{L}'$ $[(priority(l') \geq priority(l)) \wedge (l' \leq l \sqcup u)]$. This being the condition tested at line 2 of the algorithm, such an l would not have been present in $\mathcal{L}'$. Hence, we obtain the necessary contradiction. □

The proof of minimality hinges on the ability to obtain a term by instantiating variables with $\neq$. This is not always possible in a typed system since the term obtained by instantiating in this manner may violate the type discipline. Therefore, in typed systems, the above procedure computes a cover that is not necessarily minimal. In fact the following theorem shows that we are unlikely to have efficient procedures for computing a minimal cover (i.e., $\overline{\mathcal{L}_u}$) in typed systems.

THEOREM 4.2. *Computing $\overline{\mathcal{L}_u}$ is NP-complete for typed systems.*

Proof. The problem of computing $\overline{\mathcal{L}_u}$, when posed as a decision problem, takes the form "Does $l \in \overline{\mathcal{L}_u}$?" By definition of the representative set, this problem is equivalent to determining if a term t (subject to the type discipline) exists such that the following condition holds:

$$(t \geq u) \wedge (t \geq l) \wedge (\forall l' \ (priority(l') > priority(l)) \Rightarrow t \not\geq l').$$

It is easy to see that this problem is in *NP* since we need only guess a term t and check that it conforms to the type discipline and that it is an instance of u and l but does not unify with any l' of higher priority. These things can clearly be accomplished in polynomial time. To show that the problem is *NP*-complete, we will reduce the satisfiability problem (SAT [8]) to identifying such a t.

Let $\varphi = \varphi_1 \wedge \cdots \wedge \varphi_n$ be an instance of a satisfiability problem where φ_i is a disjunct of literals of the form z or $\neg z$, $z \in \{z_1, \ldots, z_m\}$. We transform this into an instance of determining whether a pattern belongs to $\overline{\mathcal{L}_u}$. Consider a (function) symbol f, taking $m + 1$ arguments and constrained by a type discipline such that each of these arguments must be either a or b. Assume that f is defined using the following $n + 1$ patterns $l_1, \ldots, l_{n+1}$ with textual order priority, i.e., $priority(l_i) > priority(l_{i+1})$ for $1 \leq i \leq n$. l_{n+1} is of the form $f(a, y_1, \ldots, y_m)$. For $1 \leq i \leq n$, $l_i = f(y_0, s_1^i, \ldots, s_m^i)$, where s_j^i for $1 \leq j \leq m$ is given by

$$s_j^i = a, \quad \text{if } z_i \text{ appears in } \varphi_i, \tag{5}$$

$$s_j^i = b, \quad \text{if } \neg z_i \text{ appears in } \varphi_i, \tag{6}$$

$$s_j^i = \_, \quad \text{otherwise.} \tag{7}$$

Now consider the problem of checking if there is an instance of $u = f(x_0, \ldots, x_m)$ that does not unify with any of the first n patterns. (This will determine if the $(n + 1)$th pattern belongs to the representative set for the prefix $f(x_0, \ldots, x_m)$.) Suppose that there is such an instance $t = u\sigma$. Consider the following truth assignment $\mathcal{T}$ derived from t:

$$z_j = false, \quad \text{if } \sigma(x_j) = a, \tag{8}$$

$$z_j = true, \quad \text{otherwise, i.e., } \sigma(x_j) = b \text{ or } \sigma(x_j) = \_. \tag{9}$$

Since $t = u\sigma$ does not unify with l_i for $1 \leq i \leq n$, we know that there exists some s_j^i such that either $s_j^i = a \neq \sigma(x_j) = b$, or $s_j^i = b \neq \sigma(x_j) = a$. In the former case, z_j appears in φ_i (by (5)) and z_j is set to *true* (by (9)) and hence φ_i is satisfied. In the latter case, $\neg z_j$ appears in φ_i (by (6)) and z_j is set to $false$ (by (8)); hence φ_i is once again satisfied. Since the above argument holds for $1 \leq i \leq n$, we have constructed a solution for the SAT.

Now we show that whenever the input to the SAT problem is satisfiable, $l_{n+1} \in \overline{\mathcal{L}_u}$. Suppose that $\mathcal{T}$ is a truth assignment that satisfies φ. Consider the following substitution σ such that $\sigma(x_0) = a$ and for other x_j, $1 \leq j \leq m$, σ is given by

$$\sigma(x_j) = b, \quad \text{if } z_j = true \text{ in } \mathcal{T}, \tag{10}$$

$$\sigma(x_j) = a, \quad \text{otherwise.} \tag{11}$$

Since $\mathcal{T}$ satisfies φ_i for $1 \leq i \leq n$, it must be true that either there is a z_j in φ_i that is set to *true* by $\mathcal{T}$, or there is a $\neg z_j$ in φ_i such that z_j is set to *false* by $\mathcal{T}$. In the former case, it is easy to see that $s_j^i = a$ (by (5)) and $\sigma(x_j) = b$ by (10) and hence $t = u\sigma$ does not unify with l_i. In the latter case, $s_j^i = b$ and $\sigma(x_j) = a$ and so t once again does not unify with l_i. Moreover, since $\sigma(x_0) = a$, $t \geq l_{n+1}$, we can answer the question of whether $l_{n+1} \in \overline{\mathcal{L}_u}$ affirmatively. □

4.2. Index computation. Recall that an index for a prefix u is a position on its fringe that must be inspected to announce a match for any pattern in $\mathcal{L}_u$ (or $\overline{\mathcal{L}_u}$). In the absence of priorities, the indices of l are exactly those fringe positions wherein l has a nonvariable. With priorities, however, we may have to inspect positions wherein l has a variable in order to rule out a match for higher priority patterns. To identify these variable positions (that are indices) Laville [17] proposed an indirect method. In this method the prioritized patterns are first transformed into an equivalent set of unprioritized patterns, and then indices are identified

using this set. Specifically, for each pattern l, the transformation generates a set $\mathcal{M}_l$ of its instances (called *minimally extended patterns*) that are not instances of any higher-priority pattern. For typed systems, only those instances that observe the type discipline are generated. The transformed system is $\bigcup_{l \in \mathcal{L}} \mathcal{M}_l$. Now the indices with respect to the prioritized patterns are identical to those with respect to the unprioritized set $\bigcup_{l \in \mathcal{L}} \mathcal{M}_l$.

Puel and Suárez [23] developed a compact representation for the sets $\mathcal{M}_l$ based on the notion of *constrained terms*. A constrained term is of the form $\{t|\varphi\}$, where t is a term and φ is a constraint obtained by combining atomic constraints of the form $s_1 \neq s_2$ using conjunction and disjunction. (Puel and Suárez use the symbol $<>$ in place of $\neq$, and refer to it as the *incompatibility* relation.) The semantics of constrained terms can be understood by making a term t with variables denote the set $\mathcal{I}(t)$ of its instances. Then the constrained term $\{t|\varphi\}$ denotes the subset of $\mathcal{I}(t)$ that satisfies the constraint φ. For example, terms that satisfy the atomic constraint $s_1 \neq s_2$ are those in the set $\mathcal{I}(s_1) - \mathcal{I}(s_2)$. A constraint $\varphi \vee \psi$ is satisfied by all (and only) terms that satisfy either φ or ψ. Similarly, the constraint $\varphi \wedge \psi$ is satisfied by all (and only) terms that satisfy φ as well as ψ. (We assume that a collection of atomic formulas used to construct a constraint does not share variables.[7]) We will use several methods to simplify constrained terms. These methods are quite intuitive and their correctness readily follows from the above semantics. (For a more formal treatment of constrained terms, see [23].)

Using constrained terms the set $\mathcal{M}_l$ can be represented compactly as

$$\{l|(l \neq l_1) \wedge \cdots \wedge (l \neq l_k)\},$$

where $l_1, \ldots, l_k$ are all the patterns with priority greater than l. Since it is not apparent how indices can be computed when constraints are of this form, Puel and Suárez first transform this constrained term so that all the constraints are on variables in l. This yields a formula in conjunctive normal form (CNF), which is then converted to an equivalent constraint in disjunctive normal form (DNF). In the DNF form, indices can be easily picked: a variable x is an index for a pattern l if a constraint on x appears in every conjunction in the DNF. The above conversion of the constraint on a pattern l from CNF to DNF is very expensive and can take $O(|l|^n)$ time in the worst case. Therefore Puel and Suárez's algorithm, which is based on such a conversion, has exponential time complexity for *both typed and untyped systems.* Laville's algorithm is also exponential since the size of the set of minimally extended patterns can be exponential in the size of the original patterns. In contrast, we now present the first polynomial-time algorithm for untyped systems that operates directly on the original patterns.

4.3. Algorithm for index computation in untyped systems. The index for a prefix is computed in two steps. First we compute the set of indices of the prefix with respect to each of the constrained patterns in $\overline{\mathcal{L}_u}$ individually. The intersection of the sets thus computed yields the indices of the prefix with respect to $\overline{\mathcal{L}_u}$. We compute the indices of a prefix u with respect to to a single constrained pattern l as follows:

Let $l_1, l_2, \ldots, l_k$ be the patterns in $\overline{\mathcal{L}_u}$ that have priority over l and also unify with l. The following two steps specify the indices of u with respect to l:

1. Each variable position p in u such that l/p is a nonvariable.
2. Each variable position p in u such that l/p is a variable and p is the only position to be instantiated in $l \sqcup u$ to determine (or rule out) a match for some higher-priority pattern l_j. More formally, there is a term s such that $(l \sqcup u)[p \leftarrow s] = l_j$.

[7]This is to prevent having constraints such as $x \neq a(y) \wedge y \neq b$ (which shares a variable y between two constraints) that will complicate the development of the materials in the rest of the section.

We now illustrate how to use the above two steps on the following patterns (with textual order priority) and the prefix $u = f(x, y, z)$.

$$\begin{aligned} l_1 &= f(a, b, c), \\ l_2 &= f(a, \_, \_), \\ l_3 &= f(\_, \_, c). \end{aligned}$$

Observe that x, y, and z are all indices of l_1 by Step 1. The only index for l_2 is x by Step 1, since Step 2 does not yield any additional indices. This is because $l_2 \sqcup u = f(a, y, z)$, and neither y nor z is the *only* variable whose instantiation can eliminate the match for l_1. Hence they do not satisfy the conditions in rule 2. For l_3, z is an index by Step 1. In Step 2, $l_3 \sqcup u = f(x, y, c)$, and comparing it with the higher priority pattern l_2, we find that x is the only variable that needs to be instantiated to rule out a match with l_2. Therefore x is an index by Step 2. The intersection of all these positions is x which is therefore an index for u. Note that this method takes $O(nS)$ time to compute all indices.

We remark that a similar algorithm was suggested by Laville as a heuristic for fast index computation. However, the question of the power (or completeness) of the heuristic is not addressed at all. In contrast we show the following theorem.

THEOREM 4.3. *The algorithm for computing indices in untyped systems is sound and complete.*

Proof. The soundness of Step 1 is obvious. For the soundness of Step 2, note that $(l \sqcup u)[p \leftarrow s] = l_j$, and also that (by definition of a pattern match) $t \geq (l \sqcup u)$ and t does not unify with l_j. From these facts it follows that t/p does not unify with s, which means that t/p is a nonvariable.

For completeness, we need to show that if a position p in u is not selected by Steps 1 or 2 then it is not an index of $l \in \overline{\mathcal{L}_u}$. This is accomplished by giving a term t such that l matches t, yet t/p is a variable. The term t is obtained by instantiating all the variables in $l \sqcup u$, except the variable at p, by the symbol $\neq$. Let $l_1, \ldots, l_k$ be all the patterns in $\overline{\mathcal{L}_u}$ with priority higher than that of l. We now show that l matches t. Since $t \geq l$, we need only show that t does not unify with any of $l_1, \ldots, l_k$. We prove this by contradiction. Assume that t unifies with l_j. Since $\neq$ is not present in any pattern, the only way t can unify with l_j is if l_j has variables at all the fringe positions of $l \sqcup u$ except possibly p. But l_j/p cannot be a variable as $l \in \overline{\mathcal{L}_u}$ (otherwise $(l \sqcup u) \geq l_j \Rightarrow l \notin \overline{\mathcal{L}_u}$ by the condition on line 2 of algorithm *computeRepSet*). Therefore l_j/p must be a nonvariable. Now note that l_j has a variable in all fringe positions of $l \sqcup u$ except p, thereby satisfying the requirement in Step 2. This means that p would have been chosen as an index by Step 2 of the above algorithm, but it is not. Hence, we obtain the necessary contradiction. □

4.4. Index computation in typed systems. Unfortunately, computing indices for typed systems is difficult. Although the method we developed for untyped systems is sound, it is not complete for typed systems. In fact, polynomial time algorithms for computing indices in typed systems are unlikely. The intuitive reason for this complexity gap between typed and untyped systems can be explained by drawing the following analogy. Observe that, in Puel and Suárez's approach, the literals in constraints generated are all of the form $x \neq t$, where x is a variable and t an arbitrary term. Such constrained terms are analogous to Boolean formulas with only negative literals and hence are trivially satisfiable (by a truth assignment that assigns *false* to every literal). Similarly, index computation is simple in untyped systems. However, in typed systems there are implicit positive constraints introduced by the type discipline. Thus we have a constraint that contains both positive and negative literals. Such a constrained term

is analogous to a Boolean formula with both positive and negative literals and thus the SAT now becomes more difficult.

THEOREM 4.4. *Index computation for typed systems is co-NP-complete.*

Proof. The index selection problem, when posed as a decision problem, takes the form "Does u possess an index with respect to pattern set $\mathcal{L}$?" To show that this problem is in *co-NP*, we need to show that the problem of deciding whether u has no index is in *NP*. To do this, let $p_1, \ldots, p_r$ be the set of fringe nodes of u. We first guess r instances $t_1, \ldots, t_r$ of u such that t_i/p_i is a variable for $1 \leq i \leq r$. Then we verify that (at least) one pattern in $\mathcal{L}$ matches each t_i and if so we declare that u has no index. All this can clearly be accomplished in polynomial time; hence the problem of determining whether u does not possess an index is in *NP* and the index selection problem is in *co-NP*.

To show that the problem is *co-NP*-complete, we reduce the complement of satisfiability to this problem. Let $\varphi_1 \wedge \cdots \wedge \varphi_n$ be an instance of SAT where φ_i is a disjunct of literals of the form x or $\neg x$, $x \in \{x_1, \ldots, x_m\}$. We transform this into an index computation problem as follows. Consider the following system consisting of $n + 1$ patterns, with textual-order priority. The root of each pattern is labeled by an $(m + 1)$ arity function symbol f (which once again stands for the common prefix shared by all the patterns). The last m arguments of f consist of nonvariables drawn from the set $\{a, b\}$. The $(n + 1)$th pattern is of the form $f(x_0, x_1, \ldots, x_m)$. To specify the first n patterns, let $t_1, \ldots, t_n$ be terms. Now we specify the ith pattern (for $1 \leq i \leq n$) as $f(t_i, s_1, \ldots, s_m)$, where s_j is a or b depending upon whether x_j or $\neg x_j$ occurs in φ_i. If neither occur in φ_i, then s_j is a variable. Observe that the size of this pattern set is polynomial in the size of $\varphi_1, \ldots, \varphi_n$. With this construction, we will now show that determining whether $f(x_0, \ldots, x_m)$ possesses an index that is equivalent to determining whether $\varphi_1 \wedge \cdots \wedge \varphi_n$ is *not* satisfiable.

First we transform the above pattern set into a set of constrained patterns. Following the transformation the $(n + 1)$th pattern becomes

$$\{f(x_0, \ldots, x_m) | (x_0 \neq t_1 \vee \varphi_1) \wedge \cdots \wedge (x_0 \neq t_n \vee \varphi_n)\}.$$

Here we have slightly abused the notation in replacing $x_j \neq a$ by x_j and $x_j \neq b$ (i.e., $x_j = a$ by type discipline) by $\neg x_j$. We now show that x_0 is *not* an index of the above constrained term iff $\varphi_1 \wedge \cdots \wedge \varphi_n$ is satisfiable. Suppose that $\varphi_1 \wedge \cdots \wedge \varphi_n$ is satisfied by a truth assignment $\mathcal{T}$. Then each φ_i is satisfied by $\mathcal{T}$. Consider the instance of the term $f(x_0, v_1, \ldots, v_m)$, where v_i is b or a, respectively, depending upon whether $\mathcal{T}(x_i)$ is *false* or *true*. Clearly, this term is an instance of the constrained term, yet x_0 is a variable in it, which means that x_0 is *not* an index. To prove the converse, suppose that x_0 is not an index, i.e., there is an instance of the constrained term that does not instantiate x_0. Then, the substitutions for each $x_1, \ldots, x_m$ must satisfy φ_1 through φ_n. This implies that $\varphi_1 \wedge \cdots \wedge \varphi_n$ must be satisfiable, thus completing the proof. □

Since the index computation problem is very hard in general, we need to examine heuristics that can speed up the process in most cases. Two such heuristics are described in [25].

5. Space and matching time complexity. We now examine upper and lower bounds on the space and matching time complexity of adaptive tree automata for several classes of patterns. Since the traversal order itself is a parameter here, we first need to clarify what we mean by upper and lower bounds. An upper bound refers to an upper bound obtained by using the best possible traversal for a set of patterns, i.e., a traversal that minimizes space (or time, as the case may be). The rationale for this definition is that for every set of patterns there exist traversal orders that can result in the worst possible time or space complexity. Clearly, it is not interesting to talk about the upper bound on size of the automaton obtained using such a

Class of Patterns	Lower bound on space	Upper bound on space	Lower bound on time	Upper bound on time
Unambiguous, no priority	$\Omega(2^{\sqrt{\alpha}})$	$O(\prod_{i=1}^{n} \lvert l_i \rvert)$	$\Omega(\alpha)$	S
Unambiguous, with priority	$\Omega(\alpha^{n-1})$	$O(\prod_{i=1}^{n} \lvert l_i \rvert)$	$\Omega(S)$	S
Ambiguous	$\Omega(\alpha^{n-1})$	$O(\prod_{i=1}^{n} \lvert l_i \rvert)$	$\Omega(S)$	S

Notation

l_i: ith pattern
n: Number of patterns
S: Total number of nonvariable symbols in patterns
α: Average number of nonvariable symbols in patterns

FIG. 6. *Space and matching time complexity of adaptive automata.*

$$\begin{bmatrix} a & a & a & \_ & \_ & \_ & a \\ b & \_ & \_ & a & a & \_ & \_ \\ \_ & b & \_ & b & \_ & a & \_ \\ \_ & \_ & b & \_ & b & b & \_ \end{bmatrix}$$

FIG. 7. *Example matrix for* $n = 4$.

(deliberately chosen) nonoptimal traversal order. Our lower bounds refer to the lower bounds obtained for any possible traversal order.

Figure 6 summarizes our results. The proof on upper bound on space follows from the result of [24]. Since a left-to-right traversal is simply a special case of an adaptive traversal, the fact that there always exists a left-to-right traversal order with an automaton size less than $O(\prod_{i=1}^{n} |l_i|)$ implies the existence of an adaptive traversal with these bounds. We now present the details of lower bound proofs. The space bounds given in this section are all independent of the traversal order and are established using flat patterns that all have a root symbol f with arbitrarily large arity. For the purposes of building either the smallest size automaton or one that does matching in the shortest possible time, flat patterns are equivalent to a set of patterns having a common prefix u whose fringe size equals the arity of f. This is because every position within the common prefix u will be an index; hence by Theorem 3.5, an automaton of smallest size (and matching time) can be obtained by first visiting all these positions. The structure of the automaton after this prefix is examined will be identical to that obtained for flat patterns after visiting the root symbol.

5.1. Unambiguous, unprioritized patterns. Consider a set of n flat patterns from the alphabet $\{f, a, b\}$ and variables. Since all flat patterns have the same root symbol f, we need only specify the arguments. Therefore the n patterns are represented by a matrix of n rows, where the ith row lists the arguments of f in the ith pattern. Each column has at most one occurrence of a, at most one occurrence of b, and the rest are all '_'s. For each pair of patterns l and l' there is at least one column wherein l and l' have different nonvariables and so the system is unambiguous. Figure 7 shows a matrix that represents the four patterns $f(a, a, a, \_, \_, \_, a)$, $f(b, \_, \_, a, a, \_, \_)$, $f(\_, b, \_, b, \_, a, \_)$, and $f(\_, \_, b, \_, b, b, \_)$. Note that because each row in the matrix contains $O(n^2)$ elements, the size of the matrix is $O(n^3)$.

In order to simplify the proof in this case we will consider only those parts of the automaton reachable without following any $\neq$ transition. Let $P(n)$ denote an instance of a problem with n such patterns. Denote by $S(n)$ the size of the smallest automaton for matching $P(n)$. Suppose that the automaton chooses some position p (which will simply be a column in the matrix) to

$$\begin{bmatrix} a & a & \_ & \_ & \_ & a \\ b & \_ & b & \_ & a & \_ \\ \_ & b & \_ & b & b & \_ \end{bmatrix} \quad \begin{bmatrix} \_ & \_ & a & a & \_ & \_ \\ b & \_ & b & \_ & a & \_ \\ \_ & b & \_ & b & b & \_ \end{bmatrix}$$

FIG. 8. *Matrices representing $\mathcal{L}_u$ for states reached by transitions on a and b.*

inspect. Now there are two cases to consider, depending upon whether one or two patterns have a nonvariable at p.

Case 1: Column p contains only one nonvariable. It is clear in this case that on a positive transition (i.e., transition on inspecting an a or b), we will again be left with the same $\mathcal{L}_u$ without any reduction in the problem size. This is because the resultant problem is represented by the matrix obtained by deleting column p from the original matrix. The matrix so obtained still represents a problem of size n. For instance, in Fig. 7, if we choose column 7 for inspection then we are left with the problem of building an automaton to match on the basis of the first six positions of each pattern. In other words, we are left with a matrix obtained by deleting the last column of Fig. 7 and hence the problem is still an instance P(4) and $S(4)$ is the size of the smallest automaton.

Case 2: Column p contains two nonvariables. In this case, based on the symbol seen in the column, we can now partition the n patterns into two sets, each consisting of $n-1$ patterns. Both these sets are represented by matrices that are obtained by deleting both the column p and one of the rows that contained a nonvariable at p. It is easy to see that each of these matrices represent $P(n-1)$ and hence the smallest matching automaton has the size $S(n-1)$. In the above example, inspecting position 2 results in the pattern sets $\{1, 2, 4\}$ and $\{2, 3, 4\}$ as shown in Fig. 8. Hence

$$S(n) = 2 * S(n-1),$$

whose solution is $\Omega(2^n)$.

Recall that the arity of f is $O(n^2)$, and that we use flat patterns in the proof to denote patterns with a common prefix u with a fringe size equal to the arity of f. In order to have a fringe size of $O(n^2)$, the prefix must have size $O(n^2)$. Thus the average size of the patterns (α) equals n^2 and so we have the following theorem.

THEOREM 5.1. *The lower bound on space required by adaptive automata for unambiguous unprioritized patterns is* $\Omega(2^{\sqrt{\alpha}})$.

5.2. Unambiguous prioritized patterns. To derive the lower bound on the size of the automaton in this case, consider the following set of flat patterns with textual order priority:

$$f(c^m(a), x_2, x_3, \ldots, x_n),\ f(x_1, c^m(a), x_3, \ldots, x_n), \ldots, f(x_1, \ldots, x_{n-1}, c^m(a)).$$

In these patterns, $c^m(a)$ is an abbreviation for the term $c(c(\cdots(c(a)\cdots)$ that contains m occurrences of c. Denote by $S(n)$ the size of the automaton for patterns of the above form. We claim that the smallest size automaton is obtained by first inspecting all the c's and then the a in the first column, then those in the second column, and so on. This is because each position examined by such an automaton is an index by Theorem 4.3. Hence, it follows by Theorem 3.5 that this automaton is no larger than any other automaton. Now consider the first m states of the automaton that correspond to inspecting all the c's in the first column. Each of these states has a transition on $\neq$ that is taken on seeing a symbol different from c. Each of these transitions leads to a state that is the root of an automaton for the remaining $n-1$ patterns. Therefore,

$$S(n) \geq mS(n-1),$$

with $S(1) = 1$. (Recall that we use breadth as the measure of size.) This means $S(n) = \Omega(m^{n-1})$. As for sufficiently large m, $m = O(\alpha)$. So, we conclude with the following theorem.

THEOREM 5.2. *The lower bound on space required by adaptive automata for unambiguous prioritized patterns is* $\Omega(\alpha^{n-1})$.

5.3. Ambiguous patterns. Consider again the set of patterns used in §5.2. Now assume that there is no priority among these patterns. Because all patterns match $f(c^m(a), c^m(a), \ldots, c^m(a))$ the system is ambiguous. Observe that the automaton A for this set of patterns must report all matches since there is no priority relationship among the patterns. We now show how we can obtain an automaton A' from A for matching the prioritized system described in §5.2. To obtain A' we simply change the *match* annotation on the final states of A so that A' announces a match for the pattern with the highest priority among those for which a match is declared by A. It is easy to see that A' is an automaton for prioritized patterns and also that it is no larger than A. Therefore, by Theorem 5.2, we have the following theorem.

THEOREM 5.3. *The lower bound on space required by adaptive automata for ambiguous patterns is* $\Omega(\alpha^{n-1})$.

5.4. Matching time. We analyze the matching time of adaptive automata in this section. Herein we derive both upper and lower bounds on the matching time. We begin our discussion with the upper bound on the matching time. We would like to recall our earlier remarks regarding using $\preceq$, a partial order, as our main approach to comparing matching time of different automata. However, to give a quantitative measure of the work involved in matching, we use average path lengths of "best possible" automata in the ordering given by $\preceq$.

It is clear that an upper bound on the matching time is given by the length of the longest root-to-leaf path in the automaton. Now observe that each state in a given path inspects distinct positions and that this position must be a nonvariable position in at least one pattern. Hence, the length of the longest path can never be more than $O(S)$. (Recall that S is the sum of sizes of all patterns.) Therefore, an upper bound on the matching time is $O(S)$.

For a lower bound on matching time for unambiguous, unprioritized patterns, consider a set of *strongly sequential patterns*. For such patterns, every prefix of any pattern possesses an index. By Theorem 3.5, the automaton with the smallest time is one that examines indices in every state. It is easy to see that, in such a case, there is exactly one final state in the automaton corresponding to a match for each pattern. Moreover, all and only the nonvariable symbols in a pattern are visited on the path to a final state announcing a match for it. Therefore, the average path length of this automaton (which is our measure of matching time) is no less than the average over the sizes of the patterns, i.e., $\Omega(\alpha)$.

In the case of prioritized or ambiguous patterns, the lower bound can be tightened. This is based on the following observations. In case of prioritized patterns, we must eliminate the possibility for a match for higher-priority patterns before looking for a match for a lower-priority pattern. This means paths leading to matches for a lower-priority pattern can be substantially longer than the pattern size. Similarly, for ambiguous patterns, the need to announce all matching patterns can make lengths of matching paths for a pattern larger than the size of the pattern being matched.

We first consider the case of unambiguous prioritized patterns. We use the example presented in §5.2 to derive the lower bound in this case. Observe that every position inspected by the automaton constructed in §5.2 is an index. This means that, by Theorem 3.5, the matching time for this automaton is the smallest. Hence the lower bound on matching time is given by the average path length of this automaton. We compute this quantity as follows. Let $T(n)$ denote the sum of lengths of all root-to-leaf paths in the automaton. Observe that each of the first m states that inspect the c's in the first pattern has a $\neq$ branch. These branches

lead to a subautomaton that matches the remaining $n-1$ patterns and hence the sum of path lengths in each of these automata is given by $T(n-1)$. For the subautomaton S_i on the ith $\neq$ branch, the sum of path lengths from root to leaves in S_i is given by

$$T(n-1) + i * (\text{number of final states in } S_i).$$

The second term in the above expression accounts for the fact that the length i from the root of the automaton to the root of S_i is added to the length of every path to a final state in S_i. Noting that the number of final states (which is same as the breadth or space complexity) in S_i is $O(m^{n-2})$, and summing the path lengths over all the subautomata and the state announcing a match for the first pattern, we have

$$T(n) = m + \Sigma_{i=1}^{m}(T(n-1) + i * O(m^{n-2})) = m + mT(n-1) + O(m^n),$$

with $T(1) = m$. It can be easily checked by substitution that $T(n)$ is $\Omega(nm^n)$. Thus the average path length given by $T(n)/O(m^{n-1})$ is $\Omega(mn) = \Omega(S)$.

The lower bound on matching time for ambiguous patterns can be obtained from that of the unambiguous prioritized patterns using arguments similar to that found in §5.3. It is quite straightforward to apply the arguments found in the proof of size complexity to matching time complexity. In particular, we can construct an automaton for prioritized patterns from that obtained for ambiguous, unprioritized patterns. This construction ensures that the matching time complexity for ambiguous patterns can never be smaller than that for unambiguous prioritized patterns.

6. Minimizing space using dags. In §3, we developed several powerful techniques to reduce the space and matching time requirement of the adaptive automata. However, the lower bound results established in the previous section indicate that the size of the automaton is likely to be large. It appears (from the proofs of lower bounds) that the main reason for the exponential space requirement is the use of tree structures in representing the automaton. Lack of sharing in trees creates duplication of functionally identical subautomata and thus wastes space. A natural solution to this problem is to implement sharing with the help of dag structure (instead of tree). We develop such a solution in this section.

An obvious way to achieve sharing is to use standard FSA minimization techniques. A method based on this approach first constructs the automaton (using algorithm *Build*) and then converts it into an (optimal) dag. However, the size of the tree automaton can be exponentially larger than that of the dag automaton. For instance, the tree automaton constructed in §5.2 has exponential size, but the corresponding dag automaton is linear! Therefore, use of FSA minimization technique is bound to be inefficient. To overcome this problem we must construct the dag automaton without generating its tree structure first. This means we must identify the equivalence of two states *without even generating* the subautomata rooted at these states. Suppose we are able to identify all such pairs of equivalent states; then the optimal dag automaton can be built directly for any set of patterns. We now propose a solution to this problem for the general case of adaptive automata. This important problem of directly building an optimal automata has remained open, even in the restricted context of left-to-right traversals [9].

Central to our construction (of a dag automaton) is a technique that detects equivalent states based on the representative sets. Consider two prefixes, u_1 and u_2, that have the same representative set $\overline{\mathcal{L}_u}$. Suppose that u_1 and u_2 differ only in those positions where every pattern in $\overline{\mathcal{L}_u}$ has a variable. Since such positions are irrelevant for determining a match, these two prefixes are equivalent. On the other hand, it can also be shown that if they have different representative sets or differ in any other position then they are not equivalent. Based on this

observation, we define the relevant prefix of u as follows. Let $p_1, p_2, \ldots, p_k$ denote (all of the) positions in u such that for each p_i there is at least one pattern in $\overline{\mathcal{L}_u}$ that has a variable at p_i, and that all other patterns in $\overline{\mathcal{L}_u}$ have a variable either at p_i or above it. The relevant prefix of u is then

$$u[p_1 \leftarrow \neq][p_2 \leftarrow \neq] \cdots [p_k \leftarrow \neq].$$

For instance, the prefixes corresponding to different states marked 's' in Fig. 3 are different, but they all have the same relevant prefix $f(x, \neq, b)$. By showing that two states are equivalent iff the corresponding relevant prefixes are identical we establish the following theorem.

THEOREM 6.1. *The automaton obtained by merging states with identical relevant prefixes is optimal.*

Proof. First we need to show that the states merged as above are indeed equivalent. Note that Condition 2 on the selection function at a state v requires that *Build* select the next position only based on portions of the prefix that are relevant for identifying a match at one of the descendents of v. This implies that the selection function will choose the same position to inspect for any two states with the same relevant prefix. It is also easy to see that if two states have the same relevant prefix, then the corresponding children of the two states will also have identical relevant prefixes. This implies that the structure of the automaton below any pair of states with the same relevant prefix will be identical. Therefore, two such states are equivalent.

Now we need to show that no two states v_1 and v_2 with distinct relevant prefixes u_1 and u_2 are equivalent. There are two cases to consider, depending upon whether $\overline{\mathcal{L}_{u_1}}$ and $\overline{\mathcal{L}_{u_2}}$ are identical or not. If they are not identical, let $l \in \overline{\mathcal{L}_{u_1}}$ and $l \notin \overline{\mathcal{L}_{u_2}}$. Then, by the properties of representative set (and the correctness of the matching automaton), there is a path from v_1 to a matching state for l. On the other hand, there is no such path from v_2 and hence v_1 and v_2 are not equivalent.

In the second case, $(\overline{\mathcal{L}_{u_1}} = \overline{\mathcal{L}_{u_2}})$. Since $u_1 \neq u_2$, $\exists p\ root(u_1/p) \neq root(u_2/p)$. Since the representative sets are the same, u_1 and u_2 cannot have different nonvariable symbols at p. Hence one of these relevant prefixes, say u_1, has a nonvariable symbol at p and the other has a variable at p. If this nonvariable is $\neq$, every pattern in the representative set must contain a variable at p. The definition of the relevant prefix then implies that $root(u_2/p)$ must also be $\neq$. Since we assumed that u_1 and u_2 differ at p, this is also not possible. Therefore $root(u_1/p)$ must be a nonvariable symbol other than $\neq$. Let l be a pattern in the representative set such that $root(l/p)$ is a nonvariable. Now there must be a path from v_1 to a matching state for l, and the symbol at p is not examined on this path (because it has already been examined in the path reaching v). In contrast, the symbol at p is examined on every path from v_2 to a matching state for l. Therefore v_1 and v_2 are not equivalent. □

Merging equivalent states as described above can substantially reduce the space required by the automata. For example, the tree automaton in Fig. 3 has 25 states which can now be reduced to 16 by sharing. Also recall that for the patterns in Fig. 7, parts of the automaton reached by positive transitions alone (i.e., without considering parts of automata that are reached through $\neq$ transitions) are exponential. We can show that by sharing states this part of the automaton will become polynomial! Similarly, the size of the automaton constructed in §5.2 for unambiguous, prioritized patterns will become linear in the size of patterns rather than being exponential. (This is because all the states that are reached on $\neq$ transitions from the first m states will be equivalent, and thus there will be only one subautomaton of size $S(n-1)$ instead of m.)

6.1. Impact of dags on space and matching time complexity. Observe that sharing affects space requirements alone. Therefore, all our earlier results not directly related to space

continue to hold for dags as well. In what follows we discuss the impact of dags on some of the results established earlier regarding space.

We can show that the upper bound on size of dag automata is $O(2^n S)$ which is much smaller than the corresponding bound $O(\prod_{i=1}^{n} |l_i|)$ for tree automata. To prove this result, consider a dag automaton based on left-to-right traversal of patterns. Consider the relevant prefixes of any two states in this automaton with the same representative set $\overline{\mathcal{L}_u}$. Since the symbols are visited in pre-order, one of these prefixes u_1 must be an instance of the other prefix u_2. Extending this argument to all states $s_1, \ldots, s_k$ with the same representative set, we see that there is a total order among these prefixes (given by the above-mentioned instance of relation). This means that the number of such prefixes (and hence the number of such states) is bounded by the size of the largest prefix, which is in turn bounded by S. This bound, in conjunction with the fact that there are at most 2^n different representative sets, yields the bound of $O(S * 2^n)$.

We can also establish a lower bound of $O(2^\alpha)$ for ambiguous patterns. For this proof, consider n flat patterns of the form

$$\begin{aligned} l_1 &= f(a, x_2, x_3, \ldots, x_n), \\ l_2 &= f(x_1, a, x_3, \ldots, x_n), \\ &\vdots \\ l_n &= f(x_1, \ldots, x_n - 1, a). \end{aligned}$$

By our earlier remarks on flat patterns, note that the average size α of these patterns is $O(n)$. There is no priority among the patterns, so the automaton is required to report all patterns that match a given term. It is clear that any term $f(t_1, \ldots, t_n)$ matches the set of patterns $\{p_{i1}, \ldots, p_{ik}\}$ whenever $t_{i1} = t_{i2} = \cdots = t_{ik} = a$. There are 2^n such sets, each of which must correspond to a state in the automaton and hence the bound.

For unambiguous patterns, it is not clear whether the lower bound on size is exponential. For instance, it appears that the patterns used in the lower bound proof on size of tree automata for unambiguous patterns possess a polynomial-size dag automaton. In the example used to establish the lower bound on space for the tree automaton for unambiguous prioritized patterns, observe that the first m states reached by $\neq$ transitions (i.e., states reached on inspecting any symbol other than c in the first column) are all equivalent. By sharing all these states, we get the recurrence relation $S(n) = m + S(n-1)$, whose solution is $O(S)$. Reasoning about lower bounds becomes extremely complicated for dags since it is difficult to capture the behavior of sharing formally.

All the greedy strategies as well as the strategy of selecting indices can, in some cases, increase the space of dag automata. This increase typically occurs in contrived examples. In practice, we should use some of the greedy strategies and the index selection strategy to reduce space and matching time. Sharing of states then provides additional opportunities for further reduction in space required.

7. Concluding remarks. In this paper we studied pattern matching with adaptive automata. We first presented a generic algorithm for their construction and then discussed how to improve space and matching time by synthesizing traversal orders. We showed that a good traversal order selects indices whenever possible and uses one of the greedy strategies otherwise. Although the greedy strategies may sometimes fail, it appears from the complexity of the counter examples that such failures may be rare. For functional programming, we synthesized a traversal $S(T)$ from a monotonic traversal T. Since using $S(T)$ does not affect

termination properties, the programmer can assume T whereas an implementation can benefit from significant improvements in space and matching time.

Our lower bound results indicate that the size of an adaptive automaton, when represented as a tree, can be quite large. Therefore, we developed an orthogonal approach to space minimization by sharing equivalent states. Note that even the index selection strategy may fail to improve the space of dag automata. This occurs because index selection may adversely affect the way in which descendants of a state can be shared. Since it is difficult to predict sharing among descendant states, the possibility of improving space without using indices does not appear to be practical. The best approach is to use all the strategies in §3 and use sharing as an additional source of space optimization over tree automata.

Our work clearly brings forth the impact of typing in prioritized pattern matching. We have shown that several important problems in the context of pattern matching are unlikely to have polynomial-time algorithms for typed systems whereas we have given polynomial-time algorithms for them in untyped systems. This raises the question whether it is worthwhile to consider typing for pattern matching. It is not clear how often typing information can be used to find an index (or to determine that a pattern does not belong to $\overline{\mathcal{L}_u}$) which cannot be found otherwise. On the other hand there is a significant penalty in terms of computational effort for both these problems if we use typing information.

We have left open two aspects of the problem that may be important in certain applications. First, our algorithms are mainly suited for an environment wherein the patterns do not change very frequently. This is because the traversal order itself can change when patterns are changed. Second, the algorithms cannot handle variables in the term to be matched. If we can handle variables, then the automaton can be used for fast unification, which has many applications such as Prolog compilers and theorem provers. We are currently investigating techniques to extend our algorithms to address these concerns.

REFERENCES

[1] L. AUGUSTSSON, *A compiler for lazy ML*, in Proc. ACM conference on LISP and Functional Programming, Austin, TX, 1984, pp. 218-225.

[2] ———, *Compiling pattern matching*, in Proc. International Conference on Functional Programming and Computer Architecture, Nancy, France, 1985. pp. 368-381.

[3] R. BURSTALL, D. MACQUEEN, AND D. SANELLA, *HOPE: An experimental applicative language*, in Proc. International LISP Conference, Stanford, CA, 1980, pp. 136-143.

[4] J. CHRISTIAN, *Flatterms, discrimination nets and fast term rewriting*, J. Automat. Reason., 10 (1993), pp. 95-113.

[5] D. COMER AND R. SETHI, *Complexity of trie index construction*, in Proc. 17th Annual IEEE Symposium on Foundations of Computer Science, Houston, TX, 1976, pp. 197-207.

[6] T. A. COOPER AND N. WOGRIN, *Rule-Based Programming with* OPS5, Morgan Kaufmann, San Francisco, 1988.

[7] N. DERSHOWITZ AND J. P. JOUANNAUD, *Rewrite systems*, in Handbook of Theoretical Computer Science, B, MIT Press, Cambridge, MA, 1990.

[8] M. GAREY AND D. JOHNSON, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.

[9] A. GRAF, *Left-to-right tree pattern matching*, in Proc. Fourth International Conference on Rewriting Techniques and Application, Como, Italy, 1991, pp. 323-334.

[10] R. HARPER, R. MILNER, AND M. TOFTE, *The definition of standard ML*, Report ECS-LFCS-88-62, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, 1988.

[11] P. HENDERSON, *Functional Programming: Application and Implementation*, Prentice-Hall, Englewood Cliffs, NJ, 1980.

[12] C. M. HOFFMANN AND M. J. O'DONNELL, *Pattern matching in trees*, J. Assoc. Comput. Mach., 29 (1982), pp. 68-95.

[13] P. HUDAK, S. L. PEYTON JONES, P. L. WADLER, B. BOUTEL, J. FAIRBURN, J. FASEL, M. GUZMAN, K. HAMMOND, J. HUGHES, T. JOHNSSON, R. KIEBURTZ, R. S. NIKHIL, W. PARTAIN, AND J. PETERSON, *Report on the functional programming language Haskell*, Version 1.2, Sigplan Notices, Section R, 27 (1992), pp. 1-154.

[14] G. HUET AND J. J. LÉVY, *Computations in nonambiguous linear term rewriting systems*, Tech. Report 359, IRIA, Le Chesney, France, 1979.
[15] G. KAHN AND G. PLOTKIN, *Domaines Concretes*, Tech. Report 336, IRIA Laboria, Le Chesney, France, 1978.
[16] J. R. KENNAWAY, *The specificity rule for lazy pattern matching in ambiguous term rewriting systems*, in Proc. European Symposium on Programming, Copenhagen, Denmark, 1990, pp. 256-270.
[17] A. LAVILLE, *Lazy pattern matching in the ML language*, in Proc. 7th Conference on Foundations of Software Technology and Theoretical Computer Science, Pune, India, 1987, pp. 400-419.
[18] ———, *Implementation of lazy pattern matching algorithms*, in Proc. European Symposium on Programming, Nancy, France, 1988, pp. 298-315.
[19] L. MARANGET, *Compiling lazy pattern matching*, in Proc. ACM Conference on Lisp and Functional Programming, San Francisco, 1992, pp. 21-31.
[20] J. MCDERMOTT AND C. L. FORGY, *Production system conflict resolution strategies*, in Pattern-Directed Inference Systems, D. Waterman and D. Hayes-Roth, eds., Academic Press, San Diego, CA, 1978, pp. 177-199.
[21] M. J. O'DONNELL, *Equational Logic as a Programming Language*, MIT Press, Cambridge, MA, 1985.
[22] K. OWEN, S. PAWAGI, C. RAMAKRISHNAN, I. V. RAMAKRISHNAN, AND R. C. SEKAR, *Fast parallel implementation of functional languages: The EQUALS experience*, in Proc. ACM Conference on Lisp and Functional Programming, San Francisco, 1992, pp. 335-344.
[23] L. PUEL AND A. SUÁREZ, *Compiling pattern matching by term decomposition*, in Proc. ACM Conference on Lisp and Functional Programming, Nice, France, 1990, pp. 273-281.
[24] PH. SCHNOEBELEN, *Refined compilation of pattern matching for functional languages*, Sci. Comput. Programming, 11 (1988), pp. 133-160.
[25] R. C. SEKAR, R. RAMESH, AND I.V. RAMAKRISHNAN, *Adaptive pattern matching*, in Proc. International Conference on Automata, Languages and Programming, Vienna, Austria, 1992, pp. 247-260.
[26] P. WADLER, *Efficient compilation of pattern matching*, in The Implementation of Functional Programming Languages, S. L. Peyton-Jones, ed., Prentice Hall, Englewood Cliffs, NJ, 1987, pp. 78-103.

SIAM J. COMPUT.
Vol. 24, No. 6, pp. 1235-1258, December 1995

007

FINDING REGULAR SIMPLE PATHS IN GRAPH DATABASES*

ALBERTO O. MENDELZON† AND PETER T. WOOD‡

Abstract. We consider the following problem: given a labelled directed graph G and a regular expression R, find all pairs of nodes connected by a simple path such that the concatenation of the labels along the path satisfies R. The problem is motivated by the observation that many recursive queries in relational databases can be expressed in this form, and by the implementation of a query language, $\mathbf{G}^+$, based on this observation. We show that the problem is in general intractable, but present an algorithm than runs in polynomial time in the size of the graph when the regular expression and the graph are free of *conflicts*. We also present a class of languages whose expressions can always be evaluated in time polynomial in the size of both the graph and the expression, and characterize syntactically the expressions for such languages.

Key words. labelled directed graphs, NP-completeness, polynomial-time algorithms, regular expressions, simple paths

AMS subject classifications. 68P, 68Q, 68R

1. Introduction.

Much of the success of the relational model of data can be attributed to its simplicity, which makes it both amenable to mathematical analysis and easy for users to comprehend. With respect to the latter, the availability of non-procedural query languages has been a great asset. However, the fact that queries which are especially useful in new application domains are not expressible in traditional query languages has led to proposals for more powerful query languages, such as the logic-based language *Datalog* [23] and our query language $\mathbf{G}^+$ [9], [10].

The original proposal for the relational model included two query languages of equivalent expressive power: the *relational calculus* and the *relational algebra* [7]. These languages have been used as the yardstick by which other query languages are classified; a query language is said to be *relationally complete* if it has (at least) the expressive power of the relational calculus. However, this notion of completeness has been questioned since it was shown that certain reasonable queries, such as finding the transitive closure of a binary relation, cannot be expressed in the calculus [3], [4]. This particular limitation is overcome in the languages $\mathbf{G}^+$ and Datalog through their ability to express recursive queries.

The design of $\mathbf{G}^+$ is based on the observation that many of the recursive queries that arise in practice—and in the literature—amount to graph traversals (for example, [1], [12], [19]). In $\mathbf{G}^+$, we view the database as a directed, labelled graph and pose queries which are graph patterns; the answer to a query is the set of subgraphs of the database that match the given pattern. Useful applications for such a language can be found in systems representing transportation networks, communication networks, hypertext documents, and so on. In our prototype implementation, queries are drawn on a workstation screen and the database and query results are also displayed pictorially.

Example 1. Let G be a graph describing a hypertext document: nodes are chunks of text and edges are links (cross-references). Readers read the document by following links. In this context, one might be interested in a query such as: Is there a way to get from section 3.1 to section 5.2 and then to the conclusion, without reading any node more than once? The corresponding $\mathbf{G}^+$ query is shown in Fig. 1. The left-hand box in the figure contains the pattern graph, while the right-hand box contains the summary graph which specifies how the

*Received by the editors September 5, 1991; accepted for publication June 27, 1994. A preliminary version of sections of this paper appeared in *Proceedings of the 15th International Conference on Very Large Data Bases*, Amsterdam, The Netherlands, August 22-25, 1989, pp. 185-193.

†Computer Systems Research Institute, University of Toronto, Toronto, Ontario M5S 1A4, Canada.

‡Department of Computer Science, University of Cape Town, Rondebosch 7700, South Africa.

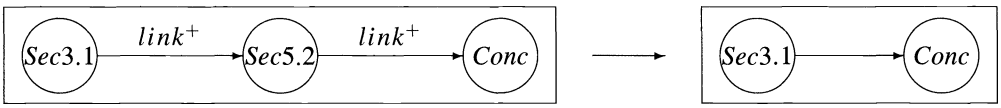


FIG. 1. *Query to test for the existence of a simple path in a hypertext document.*

FIG. 2. *Query to find pairs of cities connected by some Air Canada flight.*

output is to be presented to the user. The nodes in this case are labelled with constants to be matched with those in the database. The edges of a pattern graph can be labelled with regular expressions; in this case the desired expression is $link^+$, representing a nonzero sequence of links. This regular expression is used to match the edge labels along *simple* paths in G, thereby satisfying our original request.

Example 2. Let G be a graph representing airline flights: the nodes of G denote cities, and an edge labelled a from city b to city c means that there is a flight from b to c with airline a. Assume that we want to find all pairs of cities that are connected by a sequence of flights such that (a) at least one flight is with Air Canada (AC), and (b) no city is visited more than once. This query can be expressed by the graph pattern of Fig. 2. The pattern graph in this example comprises only two nodes, this time labelled with variables, while the edge is labelled with the regular expression $\_^*AC\_^*$ (where the underscore matches any edge label in G, and AC is regarded as a single symbol). Once again, the fact that only simple paths are matched during query evaluation ensures that the desired answer is computed.

Although queries in $\mathbf{G}^+$ can be a lot more general than those given in the above two examples, the special case suggested by Example 2 is challenging enough from an algorithmic point of view if we want to process queries efficiently. The problem addressed in this paper is, given a regular expression R and a graph G, find all pairs of nodes in G which are connected by a simple path p, where the concatenation of edge labels comprising p is in the language denoted by R.

When trying to find an efficient solution for this problem to incorporate in our implementation of $\mathbf{G}^+$, we were somewhat surprised to discover that the queries of Examples 1 and 2 are in fact both NP-complete. Using results in [11], [17], we show in §2 that for certain fixed regular expressions (such as R in Example 2), the problem of deciding whether a pair of nodes is in the answer of a query is NP-complete, making the general problem NP-hard. We first attacked this problem by determining what it is in the language of R that makes the problem hard. In §3, we present a class of languages for which query evaluation is solvable in time polynomial in both the length of the regular expression and the size of the graph. We characterize these languages syntactically in terms of the regular expressions that denote them and the finite automata that recognize them. This characterization assumes we have no knowledge concerning the structure of the graph being queried. In §4, we consider extensions where we are given a constraint which the cycles of the input graph are known to satisfy. This knowledge allows us to characterize potentially larger classes of queries which can be solved in polynomial time.

We then designed a general algorithm, presented in §5, which is correct for arbitrary graphs and queries and is guaranteed to run in polynomial time in the size of the graph if the

regular expression and graph are free of "conflicts," in a sense to be defined precisely in that section. As special cases, any query is free of conflicts with any acyclic database graph and any restricted expression query is free of conflicts with any arbitrary graph. Since we cannot restrict our prototype to work only on conflict-free queries and graphs, and it is expensive to test for conflict-freedom beforehand, it is quite convenient to have a single algorithm that works in all cases, and we have in fact incorporated the algorithm of §5 into our implementation.

2. Intractability results. In this section, we prove some negative results regarding the complexity of finding certain types of simple paths in a particular class of directed graphs. We begin by defining the graph structures as well as the class of queries over these structures in which we are interested.

DEFINITION 1. A *database graph* (*db-graph*, for short) $G = (N, E, \psi, \Sigma, \lambda)$ is a directed, labelled graph, where N is a set of *nodes*, E is a set of *edges*, and ψ is an *incidence function* mapping E to $N \times N$. Note that multiple edges between a pair of nodes are permitted in db-graphs. The labels of G are drawn from the finite set of symbols Σ, called the *alphabet*, and λ is an *edge-labelling function* mapping E to Σ.

DEFINITION 2. Let Σ be a finite alphabet disjoint from $\{\epsilon, \emptyset, (,)\}$. A *regular expression* R over Σ is defined as follows.

1. The *empty string* ϵ, the *empty set* $\emptyset$, and each $a \in \Sigma$ are regular expressions.
2. If A and B are regular expressions, then $(A + B)$, AB, and $(A)^*$ are regular expressions.
3. Nothing else is a regular expression.

The expression $(A + B)$ is called the *alternation* of A and B, (AB) is called the *concatenation* of A and B, and $(A)^*$ is called the *closure* of A. We use the underscore $(\_)$ to denote the alternation of all elements of Σ. Also, A^+ denotes AA^*, the *positive closure* of A.

The language $L(R)$ *denoted by* R is defined as follows.

1. $L(\epsilon) = \{\epsilon\}$.
2. $L(\emptyset) = \emptyset$.
3. $L(a) = \{a\}$, for $a \in \Sigma$.
4. $L(A + B) = L(A) \cup L(B) = \{w \mid w \in L(A) \text{ or } w \in L(B)\}$.
5. $L(AB) = L(A)L(B) = \{w_1 w_2 \mid w_1 \in L(A) \text{ and } w_2 \in L(B)\}$.
6. $L(A^*) = \cup_{i=0}^{\infty} L(A)^i$, where $L(A)^0 = \{\epsilon\}$ and $L(A)^i = L(A)^{i-1}L(A)$.

Regular expressions R_1 and R_2 are *equivalent*, written $R_1 \equiv R_2$, if $L(R_1) = L(R_2)$. The *length* of regular expression R, denoted $|R|$, is the number of symbols appearing in the string R.

DEFINITION 3. Let $G = (N, E, \psi, \Sigma, \lambda)$ be a db-graph and $p = (v_1, e_1, \ldots, e_{n-1}, v_n)$, where $v_i \in N$, $1 \le i \le n$, and $e_j \in E$, $1 \le j \le n - 1$, be a path (not necessarily a simple path) in G. We call the string $\lambda(e_1) \ldots \lambda(e_{n-1})$ the *path label* of p, denoted by $\lambda(p) \in \Sigma^*$. Let R be a regular expression over Σ. We say that the path p *satisfies* R if $\lambda(p) \in L(R)$. The *query* Q_R on db-graph G is defined as the set of pairs (x, y) such that there is a simple path from x to y in G which satisfies R. If $(x, y) \in Q_R(G)$, then (x, y) *satisfies* Q_R.

A naive method for evaluating a query Q_R on a db-graph G is to traverse every simple path satisfying R in G exactly once. The penalty for this is that such an algorithm takes exponential time when G has an exponential number of simple paths. Nevertheless, we will see below that in general we cannot expect an algorithm to perform much better, since we prove that, for particular regular expressions, the problem of deciding whether a pair of nodes is in the answer of a query is NP-complete. On the other hand, refinements can lead to guaranteed polynomial time evaluation under conditions studied in the following two sections.

Consider the following decision problem.

REGULAR SIMPLE PATH
Instance: db-graph $G = (N, E, \psi, \Sigma, \lambda)$, nodes $x, y \in N$, and regular expression R over Σ.
Question: Does G contain a directed simple path $p = (e_1, \ldots, e_k)$ from x to y such that p satisfies R, that is, $\lambda(e_1)\lambda(e_2)\ldots\lambda(e_k) \in L(R)$?

This is equivalent to asking "Is $(x, y) \in Q_R(G)$?" When the instance comprises only the db-graph, we refer to the problem as FIXED REGULAR PATH(R), that is, for FIXED REGULAR PATH(R) we measure the complexity only in terms of the size of the db-graph. We first prove below that, for certain regular expressions R, FIXED REGULAR PATH(R) is NP-complete. In doing so, we will refer to the following two decision problems.

EVEN PATH
Instance: Directed graph $G = (N, E)$, and nodes $x, y \in N$.
Question: Is there a directed simple path of even length (that is, with an even number of edges) from x to y?

DISJOINT PATHS
Instance: Directed graph $G = (N, E)$, and two pairs of distinct nodes $(w, x), (y, z) \in N \times N$.
Question: Is there a pair of disjoint directed simple paths in G, one from w to x and the other from y to z?

The following theorem uses the above two decision problems to prove the NP-completeness of FIXED REGULAR PATH(R) for two particular regular expressions.

THEOREM 2.1. *Let* 0 *and* 1 *be distinct symbols in* Σ. *FIXED REGULAR PATH(R), in which R is either* (1) $(00)^*$ *or* (2) 0^*10^*, *is NP-complete.*

Proof. (1) In [17], EVEN PATH is shown to be NP-complete. We can reduce EVEN PATH to FIXED REGULAR PATH(R), where $R = (00)^*$, as follows. Given an instance G, x, y of EVEN PATH, construct a db-graph H isomorphic to G, except that every edge in H is labelled with 0. There is an even simple path from x to y in G if and only if there is a simple path from x to y in H which satisfies R. It is easy to see that FIXED REGULAR PATH(R) is in NP; we conclude that FIXED REGULAR PATH(R), where $R = (00)^*$, is NP-complete.

(2) The fact that DISJOINT PATHS is NP-complete follows immediately from results in [11]. We reduce DISJOINT PATHS to FIXED REGULAR PATH(R), where $R = 0^*10^*$. Given an instance G, w, x, y, z of DISJOINT PATHS, construct a db-graph H isomorphic to G, except that every edge of H is labelled with 0. Now add a new edge (x, y) labelled 1 to H. There is a simple path from w to z satisfying R in H if and only if there are disjoint simple paths from w to x and from y to z in G. We conclude that FIXED REGULAR PATH(R), where $R = 0^*10^*$, is also NP-complete. □

COROLLARY 2.2. *REGULAR SIMPLE PATH is NP-complete.*

Proof. NP-hardness follows from Theorem 2.1. To show that REGULAR SIMPLE PATH is in NP, we observe that, for an arbitrary regular expression R, given a simple path from x to y in G with path label w, we can check in polynomial time in the lengths of R and w whether or not w is in $L(R)$ [2]. □

It is interesting to note that if G is *undirected*, then both EVEN PATH and DISJOINT PATHS can be solved in polynomial time. EVEN PATH can be solved in polynomial time by using matching techniques [17], while a polynomial-time algorithm for DISJOINT PATHS is given in [20].

Each of the two NP-completeness results of Theorem 2.1 can be generalized. We first generalize from the regular expression $(00)^*$ to expressions of the form w^*, for any $w \in \Sigma^*$ such that $|w| \geq 2$. For this we use the following NP-complete problem from [17], which was used there to show the NP-completeness of EVEN PATH.

PATH VIA A NODE
Instance: Directed graph $G = (N, E)$, and nodes $x, y, m \in N$.
Question: Is there a directed simple path from x to y via m?

THEOREM 2.3. FIXED REGULAR PATH(R), *in which* $R = w^*$, *for any* $w \in \Sigma^*$ *such that* $|w| \geq 2$, *is NP-complete.*

Proof. Once again, membership in NP is easy to demonstrate. We reduce PATH VIA A NODE to FIXED REGULAR PATH(R) using a variation of the construction from [17]. Given an instance G, x, y, m of PATH VIA A NODE, construct a db-graph $H = (N', E')$ as follows:

$$N' = ((N - \{m\}) \times \{1, 2\}) \cup \{m\},$$

$$E' = \{((u, 1), (u, 2)) \mid u \in N - \{m\}\} \tag{1}$$

$$\cup\{((u, 2), (v, 1)) \mid (u, v) \in E\} \tag{2}$$

$$\cup\{((u, 2), m) \mid (u, m) \in E\} \tag{3}$$

$$\cup\{(m, (u, 1)) \mid (m, u) \in E\}. \tag{4}$$

The proof now divides into two parts, depending on whether w is of even or odd length. Rather than introducing additional nodes into the above structure, which we believe would obscure the proof, below we allow edges to be labelled with strings of symbols. The length of a path is the length of its concatenated edge labels.

Assume that $w = w_1w_2$, where $|w_1| = n$ and $|w_2| = n$, $n \geq 1$. There are two copies of each edge of types 1 and 2 above, one copy labelled with w_1, the other with w_2. Edges of type 3 are labelled with w_2, while edges of type 4 are labelled with w_1. We claim that there is a simple path from x to y through m in G if and only if there is a simple path from $(x, 1)$ to $(y, 2)$ satisfying R in H.

If there is a path p from x to y through m in G, then let p_1 be the subpath of p from x to m, and p_2 be the subpath of p from m to y. Let u be the predecessor of m on p_1 and v be the successor of m on p_2. Then in H we can traverse a simple path from $(x, 1)$ to $(u, 2)$ which satisfies $(w_1w_2)^*w_1$, followed by the edges labelled w_2 and w_1 from $(u, 2)$ to m and from m to $(v, 1)$, respectively, followed by a simple path from $(v, 1)$ to $(y, 2)$ which satisfies $(w_2w_1)^*w_2$. The overall path thus satisfies $(w_1w_2)^*$ and is guaranteed to be simple.

Now assume there is a simple path p from $(x, 1)$ to $(y, 2)$ in H which satisfies R. All strings in $L(R)$ are of length mn, where m is even. Any path from $(x, 1)$ to $(y, 2)$ which does not pass through m must be of length kn, where k is odd. We conclude that p must pass through m in H; hence, there is a simple path from x to y via m in G.

We now consider the case in which $|w| = 2n + 1, n \geq 1$. Let $w = a_0w_1 = w_2a_{2n}$, where $|w_1| = |w_2| = 2n, n \geq 1$. One copy of each edge of type 1 in H is labelled with a_0, the other with a_{2n}. One copy of each edge of type 2 is labelled with w_1, the other with w_2. Type 3 edges are labelled with w_1, while type 4 edges are labelled with w_2.

It is easy to see that if there is a simple path from x to y via m in G, there must be a simple path satisfying R in H. For the other direction, it suffices to note that simple paths in H from $(x, 1)$ to $(y, 2)$ which do not pass through m have length $m(2n + 1) + 1, m \geq 0$, while those which do pass through m have length $k(2n + 1), k \geq 2$, which are also the lengths of strings in $L(R)$. These two can never be equal for $n \geq 0$. We conclude that if there is a simple path from $(x, 1)$ to $(y, 2)$ in H satisfying R, there must be a simple path from x to y via m in G. □

We now generalize the NP-completeness result for FIXED REGULAR PATH(R) where $R = 0^*10^*$. If $S \subseteq \Sigma$, let S also denote the alternation of its elements.

THEOREM 2.4. *Let R be a regular expression of the form S^*wT^*, where S and T are subsets of Σ and $w \in \Sigma^+$. In addition, assume that either* (1) *some a in w appears in neither S nor T, or* (2) *there are symbols $b \in S$ and $c \in T$ such that neither appears in w. Then* FIXED REGULAR PATH(R) *is NP-complete.*

Proof. Once again, FIXED REGULAR PATH(R) is obviously in NP. We use essentially the same reduction from DISJOINT PATHS to FIXED REGULAR PATH(R) as in Theorem 2.1 for this more general case.

Given an instance G, w, x, y, z of DISJOINT PATHS, construct a db-graph H isomorphic to G, except that two copies of each edge of H are made, one labelled with $b \in S$ and one labelled with $c \in T$. For case (2), b and c are those symbols mentioned in the statement of the theorem; for case (1), we choose $b \neq a$ and $c \neq a$. Assume that $w = a_1a_2 \ldots a_n$. Now add $n-1$ nodes $v_1, v_2, \ldots, v_{n-1}$ to H, along with the path $p_w = (x, e_1, v_1, \ldots, e_{n-1}, v_{n-1}, e_n, y)$, where e_i is labelled with a_i, $1 \leq i \leq n$.

If there are disjoint simple paths from w to x and from y to z in G, it is easy to see that there must be a simple path from w to z satisfying R in H. Assume now that there is a simple path p from w to z satisfying R in H. Then p must be of the form $p_1 p_w p_2$, since, in both cases (1) and (2), p_w contains an edge label which appears nowhere else in H and has to appear on any path in H satisfying R. We conclude that there must be disjoint simple paths from w to x and from y to z in H, and hence in G. □

Theorems 2.3 and 2.4 are rather negative results, since they imply that queries might require time which is exponential in the size of the db-graph, not only the regular expression, for their evaluation. Thus, for regular expressions such as those in Theorems 2.3 and 2.4, we certainly would not expect an evaluation algorithm to run in polynomial time. One such example is the "Air Canada" query used in Example 2 (as long as the alphabet Σ contains at least two symbols). These results, however, are not a function of the particular regular expression but rather of the nature of the language denoted by the regular expression. A class of languages for which REGULAR SIMPLE PATH is in P is the subject of the next section.

3. Restricted regular expressions. In this section, we characterize a class of queries about regular simple paths which can be evaluated in polynomial time. We first introduce some terminology and definitions.

DEFINITION 4. A *nondeterministic finite automaton* (NDFA) M is a 5-tuple $(S, \Sigma, \delta, s_0, F)$, where S is a finite set of *states*, Σ is the *input alphabet*, δ is the *state transition function* which maps $S \times (\Sigma \cup \{\epsilon\})$ to the set of subsets of S, $s_0 \in S$ is the *initial state*, and $F \subseteq S$ is the set of *final states*. The *extended* transition function δ^* is defined as follows. For $s, t \in S$, $a \in \Sigma$, and $w \in \Sigma^*$

$$\delta^*(s, \epsilon) = \{s\}, \text{ and}$$

$$\delta^*(s, wa) = \bigcup_{t \in \delta^*(s,w)} \delta(t, a).$$

The NDFA M *accepts* $w \in \Sigma^*$ if $\delta^*(s_0, w) \cap F \neq \emptyset$. The language $L(M)$ *accepted* by M is the set of all strings accepted by M. A *deterministic finite automaton* (DFA) is an NDFA in which the state transition function is a mapping from $S \times \Sigma$ to S.

DEFINITION 5. Let $M = (S, \Sigma, \delta, s_0, F)$ be an NDFA. The *transition graph* associated with M is a directed, labelled graph $(S, E_M, \psi_M, \Sigma, \lambda_M)$. If $t \in \delta(s, a)$ for $s, t \in S$ and $a \in \Sigma$, then there is an edge e in E_M with $\psi_M(e) = (s, t)$ and $\lambda_M(e) = a$. By confusing representations, we will sometimes say there is a *transition* from state s to state t in M (or t is a *successor* of s) if $t \in \delta(s, a)$, and there is a *path* from s to t if $t \in \delta^*(s, w)$ for some $w \in \Sigma^*$. Again, similar definitions apply for a DFA.

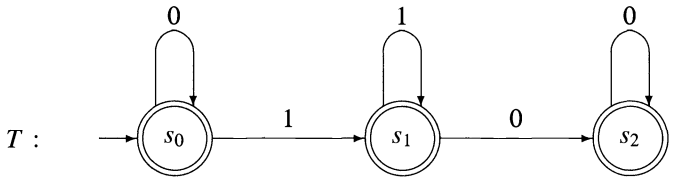


FIG. 3. *Transition graph T of a DFA.*

DEFINITION 6. Given an NDFA $M = (S, \Sigma, \delta, s_0, F)$, for each pair of states $s, t \in S$, we define the *language from s to t*, denoted by L_{st}, as the set of strings that take M from state s to state t. Then, for state s and set of states T, we can define the *language from s to T*, denoted by L_{sT}, as $\bigcup_{t \in T} L_{st}$. In particular, for a state $s \in S$, the *suffix language* of s, denoted by L_{sF} (or $[s]$, for short), is the set of strings that take M from s to some final state. Clearly, $[s_0] = L(M)$. Similar definitions apply for a DFA.

Given a regular expression R over Σ, an ϵ-free NDFA $M = (S, \Sigma, \delta, s_0, F)$ which accepts $L(R)$ can be constructed in polynomial time [2]. From now on, we will assume that all NDFAs are ϵ-free.

Example 3. Figure 3 shows the transition graph T of a DFA M. State s_0 is the initial state of M, while all states are final (denoted by a double circle). (We do not show (reject) states which are not on some path from the initial state to a final state.) $L(M)$ is denoted by the regular expression $0^*1^*0^*$. The suffix language of state s_1 is $[s_1] = 1^*0^*$, while $[s_2] = 0^*$.

Let R_1 and R_2 be regular expressions. In the subsequent analysis, it will be useful to refer to an NDFA which accepts the language $L(R_1 \cap R_2)$. The construction of such an NDFA is defined as follows.

DEFINITION 7. Let $M_1 = (S_1, \Sigma, \delta_1, p_0, F_1)$ and $M_2 = (S_2, \Sigma, \delta_2, q_0, F_2)$ be NDFAs. The NDFA for $M_1 \cap M_2$ is $I = (S_1 \times S_2, \Sigma, \delta, (p_0, q_0), F_1 \times F_2)$, where, for $a \in \Sigma$, $(p_2, q_2) \in \delta((p_1, q_1), a)$ if and only if $p_2 \in \delta_1(p_1, a)$ and $q_2 \in \delta_2(q_1, a)$. We call the transition graph of I the *intersection graph* of M_1 and M_2.

We saw in the previous section that, for certain regular expressions R, it is very unlikely that we will find an algorithm for evaluating Q_R on an arbitrary graph G which will always run in time polynomial in the size of G. One such regular expression is 0^*10^*. However, it turns out that if the regular expression $R = 0^*10^* + 0^*$ is specified instead, then Q_R is evaluable in polynomial time on any db-graph G. The reason is that if there is an arbitrary path from node x to node y in G which satisfies R, then there is a simple path from x to y satisfying R. In such a case, we need not restrict ourselves to looking only for simple paths in G, but can instead look for *any* path satisfying R. We define the corresponding decision problem below.

REGULAR PATH
Instance: db-graph $G = (N, E, \psi, \Sigma, \lambda)$, nodes $x, y \in N$, and regular expression R over Σ.
Question: Does G contain a directed path (not necessarily simple) $p = (e_1, \ldots, e_k)$ from x to y such that p satisfies R, that is, $\lambda(e_1)\lambda(e_2)\cdots\lambda(e_k) \in L(R)$?

LEMMA 3.1. *REGULAR PATH can be decided in polynomial time.*

Proof. Given db-graph G along with nodes x and y in G, we can view G as an NDFA with initial state x and final state y. Construct the intersection graph I of G and $M = (S, \Sigma, \delta, s_0, F)$, an NDFA accepting $L(R)$. There is a path from x to y satisfying R if and only if there is a path in I from (x, s_0) to (y, s_f), for some $s_f \in F$. All this can be done in polynomial time [14]. □

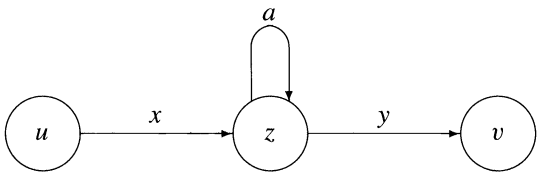


FIG. 4. *A graph containing a nonsimple path.*

In [22], Tarjan provides a polynomial-time algorithm for constructing a regular expression which represents the set of all paths between two nodes of a given graph. As an alternative to the above procedure, one could decide in polynomial time whether there was a path between x and y in G satisfying R by first using Tarjan's algorithm to construct a regular expression R_{xy} representing all paths between x and y in G, and then determining whether the intersection of $L(R)$ and $L(R_{xy})$ was nonempty using NDFAs. The results of the previous section show that it is unlikely that a polynomial-time analogue of Tarjan's algorithm exists for describing the set of all *simple* paths between two nodes.

DEFINITION 8. Let G be a db-graph, $M = (S, \Sigma, \delta, s_0, F)$ a DFA or NDFA, and I the intersection graph of G and M. We call a node (x, s_0) in I an *initial* node, and a node (y, s_f), $s_f \in F$, a *final* node.

We are interested in conditions under which REGULAR SIMPLE PATH (which is appropriate because of our semantics) can be reduced to REGULAR PATH. The following lemma states one such condition.

LEMMA 3.2. *REGULAR SIMPLE PATH can be decided in polynomial time on acyclic db-graphs.*

Proof. The proof follows immediately from Lemma 3.1 and the fact that every path in an acyclic graph is simple. □

Suppose that we want to characterize a class of regular expressions for which we can guarantee that REGULAR SIMPLE PATH is solvable in polynomial time. If we assume that we know nothing about the structure of the db-graphs, we have to ensure that, for such a regular expression R, whenever string w is in $L(R)$, every string obtainable from w by removing one or more symbols must also be in $L(R)$. Otherwise, if $w = xay$ is in $L(R)$ but xy is not in $L(R)$ (where $a \in \Sigma$ and $x, y \in \Sigma^*$), we can construct a graph G comprising a single simple path from u to v and passing through z in which there is a loop at z labelled a, the path from u to z is labelled x, and the path from z to v is labelled y (see Fig. 4). There is a nonsimple path from u to v in G which satisfies R but no simple path from u to v satisfying R.

DEFINITION 9. An *abbreviation* of a string w is any string which can be obtained from w by removing one or more symbols of w [6].

So we are looking for a class of regular expressions which denote languages that are closed under abbreviation. Now consider the following definition for the class of restricted regular expressions.

DEFINITION 10. For $a \in \Sigma$, denote the regular expression $(a + \epsilon)$ by $(a?)$ (as is done in the *grep* utility of Unix,[1] for example). Given a regular expression R, let R' be the regular expression obtained by replacing some occurrence of a symbol $a \in \Sigma$ in R by $(a?)$. R is *restricted* if and only if $R \equiv R'$, for any R' obtained from R as defined above.

Note that the above definition of restricted regular expressions is semantic rather than syntactic. This has two significant consequences: on the one hand, we are able to prove an equivalence theorem below (Theorem 3.4) relating restricted regular expressions to languages

[1] Unix is a trademark of AT&T.

and automata; on the other, the recognition problem for restricted regular expressions becomes difficult (Corollary 3.10).

Example 4. The regular expression $0^*1^*0^*$ is restricted: it is equivalent to $(0?)^*(1?)^*(0?)^*$. Recall from Theorem 2.1 that FIXED REGULAR PATH(R) is NP-complete for $R = 0^*10^*$. R is not restricted, but $R' = 0^*10^* + 0^*$ is restricted, since R' can be written as $0^*(1 + \epsilon)0^*$, which is equivalent to $(0?)^*(1?)(0?)^*$.

DEFINITION 11. A DFA $M = (S, \Sigma, \delta, s_0, F)$ exhibits the *suffix language containment property* (the *containment property*, for short) if, for each pair $s, t \in S$ such that s and t are on a path from s_0 to some final state and t is a successor of s, $[s] \supseteq [t]$ (that is, $L_{sF} \supseteq L_{tF}$).

The following result, although not used elsewhere, provides some interesting restrictions on the structure of DFAs that exhibit the containment property.

PROPOSITION 3.3. *Let $M = (S, \Sigma, \delta, s_0, F)$ be a DFA. If M exhibits the containment property, then*

1. *every state in M, which is on a path from s_0 to a state in F, is final,*
2. *the minimum DFA for M exhibits the containment property, and*
3. *if M is minimum, then every cycle in M is a loop.*

Proof. (1) Every final state in M accepts ϵ. By the transitivity of "$\supseteq$", every state which is on a path from s_0 to a state in F must also accept ϵ, and hence must be final.

(2) Let $M' = (S', \Sigma, \delta', s_0, F')$ be the minimum DFA equivalent to M. Each state in M' represents a set of equivalent states in M. Assume that $s \in S'$ represents $\{s_1, \ldots, s_k\}$, where $s_i \in S$, $1 \le i \le k$, and that $t \in S'$ represents $\{t_1, \ldots, t_m\}$, where $t_j \in S$, $1 \le j \le m$. There is a transition $\delta'(s, a) = t$ in M' only if, for each s_i, $1 \le i \le k$, in M, there is a transition $\delta(s_i, a) = t_{j_i}$, for some $1 \le j_i \le m$. In M, $[s_i] \supseteq [t_{j_i}]$, $1 \le j_i \le m$, $1 \le i \le k$. Since M' is equivalent to M, $[s] = [s_i]$, $1 \le i \le k$, and $[t] = [t_{j_i}]$, $1 \le j_i \le m$, $1 \le i \le k$. We conclude that $[s] \supseteq [t]$.

(3) Consider a cycle in M which is not a loop, and let s and t be two states on the cycle. Since $[u] \supseteq [v]$ for every pair of consecutive states on the cycle, we conclude from the transitivity of "$\supseteq$" that $[s] \supseteq [t]$ and that $[t] \supseteq [s]$. But then $s \equiv t$, and so M is not minimum, a contradiction. □

Example 5. Consider the regular expression $R = 0^*1^*0^*$, and the DFA M accepting $L(R)$ whose transition graph T is given in Fig. 3. We can verify that M exhibits the containment property by noting that $[s_2]$ is denoted by 0^*, $[s_1]$ by 1^*0^*, and $[s_0]$ by $0^*1^*0^*$. Obviously, $[s_0] \supseteq [s_0]$, $[s_1] \supseteq [s_1]$, and $[s_2] \supseteq [s_2]$. It is easy to check that $[s_1] \supseteq [s_2]$ and $[s_0] \supseteq [s_1]$. Note also that, by Proposition 3.3, each state is final and, since M is minimal, every cycle in M is a loop. The fact that M exhibits the containment property and R is restricted is no coincidence, as we demonstrate below.

THEOREM 3.4. *Let R be a regular expression over Σ, and $M = (S, \Sigma, \delta, s_0, F)$ be a DFA accepting $L(R)$. The following three statements are equivalent:*

1. *R is a restricted regular expression,*
2. *$L(R)$ is closed under abbreviations, and*
3. *M exhibits the containment property.*

Proof. In our proof, we will use the NDFA $M_R = (T, \Sigma, \mu, t_0, E)$ constructed from regular expression R (such that $L(M_R) = L(R)$) as detailed in [2], and in which ϵ-transitions are usually present. There is a one-to-one correspondence between non-ϵ-transitions in M_R and occurrences of symbols in R, so that it makes sense to refer to *the* transition in M_R corresponding to an occurrence of symbol a in R, and vice versa. Furthermore, replacing an occurrence of a in R by $(a?)$ is equivalent to including an ϵ-transition from the source state to the target state of the transition in M_R corresponding to the occurrence of a.

(1) $\Rightarrow$ (2) Assume that R is restricted but that $L(R)$ is not closed under abbreviations. Then there is a symbol $a \in \Sigma$ and strings $x, y \in \Sigma^*$ such that $xay \in L(R)$ but $xy \notin L(R)$. Now consider M_R. Let $T' = \mu^*(t_0, x)$, that is, the set of states M_R can be in after reading x. Since $L(M_R) = L(R)$ and $xy \notin L(R)$, for no $r \in T'$ can y be in $[r]$. On the other hand, $xay \in L(M_R)$, so there is a state $p \in T'$ such that $q \in \mu(p, a)$ and $y \in [q]$. Since R is restricted, adding an ϵ-transition from p to q leaves $L(M_R)$ unchanged. But if we do so, then $y \in [p]$, $xy \in L(M_R)$, and $L(M_R)$ is no longer equal to $L(R)$, which is a contradiction. We conclude that $L(R)$ is closed under abbreviations.

(2) $\Rightarrow$ (3) We prove the contrapositive. Assume that $[s] \not\supseteq [t]$ for some pair s, t of reachable states in M such that $\delta(s, a) = t$, for some $a \in \Sigma$. That is, there is a string $y \in \Sigma^*$ for which $y \in [t]$ but $y \notin [s]$. Let $x \in \Sigma^*$ be a string for which $\delta^*(s_0, x) = s$. It follows that $xay \in L(M)$, but that $xy \notin L(M)$. Since $L(M) = L(R)$, we conclude that $L(R)$ is not closed under abbreviations.

(3) $\Rightarrow$ (1) Once again we prove the contrapositive. Assume that R is not restricted. Then there is an a-transition in M_R from s to t for which adding an ϵ-transition from s to t alters $L(M_R)$. Let $x \in \Sigma^*$ be a string for which $s \in \mu^*(t_0, x)$. That is, there is a string $y \in [t]$ such that $y \notin [r]$ for any $r \in \mu^*(t_0, x)$; hence, $xy \notin L(M_R)$. Now consider the DFA M. Assume that $\delta^*(s_0, x) = p$. Since $L(M_R) = L(M)$ and $xay \in L(M_R)$, there must be a state q in M such that $\delta(p, a) = q$ and $y \in [q]$. However, $y \notin [p]$, for otherwise $xy \in L(M)$ which would mean that $L(M) = L(M_R)$. Hence, $[p] \not\supseteq [q]$, so M does not exhibit the containment property. □

THEOREM 3.5. *REGULAR SIMPLE PATH can be decided in polynomial time for restricted regular expressions.*

Proof. Let the db-graph G and the regular expression R, where R is restricted, constitute an instance of REGULAR SIMPLE PATH. By Lemma 3.1, it is sufficient to show that whenever there is a path from x to y in G which satisfies R, there is a simple path from x to y satisfying R. Assume that $p = (v_1, e_1, \ldots, e_{n-1}, v_n)$ is a nonsimple path from $x = v_1$ to $y = v_n$ in G. Since p is nonsimple, $v_i = v_j$, for some $1 \leq i, j \leq n$. Assume that $i < j$, that is, $p = (v_1, \ldots, e_{i-1}, v_i, \ldots, e_{j-1}, v_i, e_j, \ldots, v_n)$, and let $p' = (v_1, \ldots, e_{i-1}, v_i, e_j, \ldots, v_n)$. Since p satisfies R, $\lambda(p) \in L(R)$. The path label $\lambda(p')$ is an abbreviation of $\lambda(p)$. By Theorem 3.4, $L(R)$ is closed under abbreviations; hence, $\lambda(p') \in L(R)$ and p' satisfies R. Removing all such cycles from p will leave a simple path from x to y which satisfies R. □

Thus the class of restricted regular expressions is one for which query evaluation can be performed efficiently. We now show that, even though the classes of restricted regular expressions and regular languages closed under abbreviations are subclasses of their regular counterparts, at least they are closed under the regular operators.

THEOREM 3.6. *Let Σ be an alphabet. The class of regular languages over Σ which is closed under abbreviations is also closed under alternation, concatenation, and closure.*

Proof. Let L_1 and L_2 be regular languages closed under abbreviations. It is immediate that $L_1 + L_2$ is closed under abbreviations too. Now let $L = L_1 L_2$ and consider $w = w_1 w_2 \in L$ such that $w_1 \in L_1$ and $w_2 \in L_2$. Let $w' = w_1' w_2'$ be an abbreviation of w. Clearly, string w_i' is an abbreviation of w_i, $i = 1, 2$, and since L_1 and L_2 are closed under abbreviations, $w_1' \in L_1$ and $w_2' \in L_2$. Hence, $w_1' w_2' = w'$ is in L, allowing us to conclude that L is closed under abbreviations.

Let L be a regular language closed under abbreviations. Since $\epsilon \in L$ and regular languages closed under abbreviations are also closed under concatenation, L^* must be closed under abbreviations. □

COROLLARY 3.7. *The class of restricted regular expressions over Σ is closed under alternation, concatenation, and closure.*

Algorithm S: Compute the suffix language containment relation for a DFA.

INPUT:
DFA $M = (S, \Sigma, \delta, s_0, F)$

OUTPUT:
For each pair $s, t \in S$, whether $[s] \supseteq [t]$ or not.

METHOD:

```
1.    for s ∈ S − F and t ∈ F do mark (s, t) od
2.    for each ordered pair of distinct states (s, t) ∈ ((S × S) − ((S − F) × F)) do
3.        if for some a ∈ Σ (δ(s, a), δ(t, a)) is marked then
4.            mark (s, t)
5.            recursively mark all unmarked pairs on the list for (s, t) and
              on the lists of other pairs that are marked at this step
          else /* no pair (δ(s, a), δ(t, a)) is marked */
6.            for all a ∈ Σ do
7.                put (s, t) on the list for (δ(s, a), δ(t, a)) unless δ(s, a) = δ(t, a)
              od
          fi
      od
```

FIG. 5. *Computing the suffix language containment relation for DFA* $M = (S, \Sigma, \delta, s_0, F)$.

Example 6. One of the simplest restricted regular expressions is 0^*. Since the class of restricted regular expressions is closed under alternation, concatenation, and closure, $0^* + 1^*$ and $0^*1^*0^*$ (which we have already seen) are restricted. On the other hand, restricted expressions can also sometimes be built from expressions which are not restricted; examples include $0^*10^* + 0^*$ (which we have already seen), $(00)^* + 0^*$, and $((0^*1)^* + 0^*)^*$.

Given a query Q_R, we would like to test whether R is restricted in order to know that it is safe to use a polynomial time evaluation algorithm. By adapting an algorithm to minimize the number of states of a DFA [13], we can compute the suffix language containment relation for all pairs of states in a DFA M. The suffix language containment relation will be used in subsequent sections; it also provides an obvious method for testing whether or not a regular expression R is restricted (using Theorem 3.4). The algorithm for computing the suffix language containment relation, Algorithm S, is shown in Fig. 5. Lines 3 to 7 of Algorithm S are taken directly from the algorithm in [13]. That algorithm marks pairs of *inequivalent* states, so it considers unordered pairs of states. Lines 1 and 2 of our algorithm are altered appropriately in order to consider ordered pairs of states. If (s, t) is marked by Algorithm S, then $[s] \not\supseteq [t]$.

If M has n states, then Algorithm S runs in $O(n^2)$ time (assuming a constant alphabet) [13]. (An alternative, almost linear-time algorithm is given in [2].) Since the construction of a DFA M accepting $L(R)$ may take exponential time (in the size of R), using Algorithm S to test whether a regular expression is restricted is not efficient. However, it is important to stress that we are trying to avoid the possibility of spending exponential time in the size of the db-graph in answering a query. Also, it turns out that determining whether or not R is restricted is a hard problem. Consider the following result.

PROPOSITION 3.8 [21]. *Determining whether a regular expression over alphabet* $\{0\}$ *does not denote* 0^* *is NP-complete.*

We will use this result to show that the problem of deciding whether a regular expression over alphabet Σ is not restricted is NP-hard. To do so, we first prove the following.

THEOREM 3.9. *Let R be a starred regular expression over alphabet $\{0\}$. Deciding whether R is not restricted is NP-complete.*

Proof. We first show that the problem is in NP. If R is not restricted, then $L(R)$ is not closed under abbreviations (Theorem 3.4). Thus, there is a string in 0^* that is not in $L(R)$. If $L(R) \neq 0^*$, then, by considering a DFA accepting $L(R)$, it can be seen that there must be an $n \leq 2^{|R|}$ such that $0^n \notin L(R)$. A nondeterministic polynomial time algorithm can verify that R is not restricted by first guessing the binary representation of n and then testing whether there is a path in the transition graph of an NDFA accepting $L(R)$ of length n to a final state. The latter step can be done deterministically in time polynomial in the length of R [21].

We reduce the problem of Proposition 3.8 to the present problem by showing that R is not restricted if and only if R does not denote 0^*. We have already shown that if R is not restricted, then $L(R) \neq 0^*$. Conversely, assume that R does not denote 0^*. Let x be the shortest string in 0^* that is not in $L(R)$. Since R is starred, $L(R)$ is infinite, so there is a string $xy \in L(R)$ for which $y \neq \epsilon$. But x is an abbreviation of xy; hence, by Theorem 3.4, R is not restricted. □

COROLLARY 3.10. *Deciding whether a regular expression over alphabet Σ is not restricted is NP-hard.*

4. Constrained cycles in db-graphs. In some instances, knowledge about the cyclic structure of a db-graph G allows us to determine (without consulting G itself) that a particular query Q_R can be evaluated in polynomial time on G. We have already shown that, in the extreme case when G is acyclic, Q_R is always evaluable in polynomial time. Let us assume that we know that the cyclic structure of G is constrained by a regular expression C; that is, every cycle label in G is in $L(C)$.

DEFINITION 12. Let C be a regular expression over Σ, and $G = (N, E, \psi, \Sigma, \lambda)$ be a db-graph. Let Y be the set of cycle labels in G, namely

$$Y = \{\lambda(c) \mid c \text{ is a cycle in } G\}.$$

We say that G *complies with* C if $Y \subseteq L(C)$. The regular expression C is called a *cycle constraint.*

Each cycle constraint C defines a class of db-graphs whose cyclic structure satisfies C. For example, in this way we can define the classes of bipartite graphs, loop-free graphs, and acyclic graphs by specifying the regular expressions $(\_\ \_)^+$, $\_\ \_(\_^*)$, and $\emptyset$, respectively.[2] The class of db-graphs with unconstrained cycles is defined by the expression $\_^+$, which denotes Σ^+.

Before continuing, we need to introduce some terminology regarding properties of the intersection graph of a db-graph and a transition graph.

DEFINITION 13. Let I be the intersection graph of db-graph $G = (N, E, \psi, \Sigma, \lambda)$ and transition graph T of NDFA $M = (S, \Sigma, \delta, s_0, F)$. We say that a path $p = ((v_1, s_1), \ldots, (v_n, s_n))$, where $v_i \in N$ and $s_i \in S$, in I is *db-simple* if $v_i \neq v_j$, $1 \leq i, j \leq n$. In other words, p is db-simple if and only if $(v_1, \ldots, v_n)$ is a simple path in G. In addition, we call I *simplicial* if whenever there is a path $p = ((v_1, s_1), \ldots, (v_n, s_n))$, where $v_1 \neq v_n$ and $s_n \in F$, there is a db-simple path from (v_1, s_1) to (v_n, s_n'), $s_n' \in F$, in which the first components of nodes form a subset of the first components of nodes on p.

From the above definition and Lemma 3.1, it is clear that if the intersection graph I of a db-graph and the transition graph corresponding to a regular expression R is simplicial, then Q_R can be evaluated in polynomial time in the size of I. The following theorem characterizes simplicial intersection graphs in the presence of cycle constraints.

[2]Recall that if $\Sigma = \{a_1, \ldots, a_n\}$, then _ (underscore) is shorthand for $a_1 + \cdots + a_n$.

THEOREM 4.1. *Let C be a cycle constraint. For query Q_R, let $M = (S, \Sigma, \delta, s_0, F)$ be a DFA accepting $L(R)$ and T be the transition graph of M. For every db-graph G complying with C, the intersection graph I of G and T is simplicial if and only if whenever there is a path from a reachable state s to t in T satisfying C, $[s] \supseteq [t]$.*

Proof. (If) Let $G = (N, E, \psi, \Sigma, \lambda)$ be a db-graph complying with C and

$$p = (v_1, \ldots, e_{i-1}, v_i, \ldots, e_{j-1}, v_i, \ldots, v_n)$$

be a nonsimple path satisfying R in G. Hence, there is a path q from (v_1, s_0) to (v_n, s_f), $s_f \in F$, in I. For notational simplicity, let $w_1 = \lambda(e_1) \ldots \lambda(e_{i-1})$, $w_2 = \lambda(e_i) \ldots \lambda(e_{j-1})$, and $w_3 = \lambda(e_j) \ldots \lambda(e_{n-1})$. So $w_1 w_2 w_3 \in L(R)$. Since G complies with C, $w_2 \in L(C)$. Assume that $\delta^*(s_0, w_1) = s$ and $\delta^*(s, w_2) = t$. So there is a path in T from s to t satisfying C and a path from (v_i, s) to (v_i, t) in I; hence, by assumption, $[s] \supseteq [t]$. The string w_3 is in $[t]$ because p satisfies R, so $w_3 \in [s]$ as well. It follows that $w_1 w_3 \in L(R)$ and therefore that

$$p' = (v_1, \ldots, e_{i-1}, v_i, e_{j+1}, \ldots, v_n)$$

satisfies R. This process can be repeated to obtain a db-simple path q' from (v_1, s_0) to (v_n, s'_f), $s'_f \in F$, such that the first components of q' form a subset of the first components of q. We conclude that I is simplicial.

(Only if) Assume that there is a path p from s to t in T which satisfies C but for which $[s] \not\supseteq [t]$. The constraint C cannot be $\emptyset$, for otherwise p would not satisfy C. Since s is reachable in T, there is a string w_1 such that $\delta^*(s_0, w_1) = s$. Furthermore, $[t]$ cannot be $\emptyset$, for otherwise $[s] \supseteq [t]$. So let w_3 be a string in $[t]$ but not in $[s]$, and w_2 be the path label of p. The string w_2 cannot be ϵ since p must be of length greater than zero. We can construct a db-graph $G = (N, E, \psi, \Sigma, \lambda)$ comprising a single nonsimple path

$$q = (v_1, \ldots, e_{i-1}, v_i, \ldots, e_{j-1}, v_i, \ldots, v_n)$$

such that $\lambda(e_1) \ldots \lambda(e_{i-1}) = w_1$, $\lambda(e_i) \ldots \lambda(e_{j-1}) = w_2$, and $\lambda(e_j) \ldots \lambda(e_{n-1}) = w_3$. G complies with C since the only cycle in G is labelled with w_2 which is in $L(C)$. The path q satisfies R because $w_1 w_2 w_3 \in L(R)$. Hence, there is a path from (v_1, s_0) to (v_n, s_f), $s_f \in F$, in I. However, the path

$$q' = (v_1, \ldots, e_{i-1}, v_i, e_{j+1}, \ldots, v_n)$$

does not satisfy R since $w_1 w_3 \notin L(R)$ (otherwise w_3 would be in $[s]$). Consequently, there is no db-simple path from (v_1, s_0) to (v_n, s'_f), $s'_f \in F$, in I, and we conclude that I is not simplicial. □

The above result does not depend on the particular DFA accepting $L(R)$. Consider two DFAs, $M_1 = (S_1, \Sigma, \delta_1, s_0, F_1)$ and $M_2 = (S_2, \Sigma, \delta_2, t_0, F_2)$, accepting $L(R)$, and let $s \in S_1$ and $t \in S_2$ be a pair of states such that there is a string x for which $\delta_1^*(s_0, x) = s$ and $\delta_2^*(t_0, x) = t$. Because $L(M_1) = L(M_2)$, it must be the case that $[s] = [t]$ (that is, $s \equiv t$). In other words, the fact that Theorem 4.1 is true independent of the particular DFA chosen is a consequence of the Myhill–Nerode theorem, which states that a language is accepted by a DFA if and only if it is the union of some of the equivalence classes of a right-invariant equivalence relation of finite index [13]. This leads us to the following definition.

DEFINITION 14. Let R be a regular expression and T be the transition graph for a DFA accepting $L(R)$. We say that R is *compatible with* cycle constraint C if whenever there is a path from (a reachable state) s to t in T satisfying C, $[s] \supseteq [t]$.

Theorem 4.1 generalizes our previous results. For the case when G is acyclic, $C = \emptyset$, and no path in T satisfies C so the result holds vacuously. In other words, every regular expression

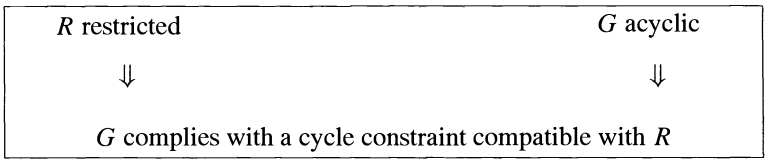


FIG. 6. *Relationship between regular expression R and db-graph G for query Q_R.*

Algorithm: Testing whether a regular expression is compatible with a cycle constraint.

INPUT:
Regular expression R and cycle constraint C.

OUTPUT:
Whether or not R is compatible with C.

METHOD:
1. Construct DFAs $M_R = (S_1, \Sigma, \delta_1, i_1, F_1)$ accepting $L(R)$ and $M_C = (S_2, \Sigma, \delta_2, i_2, F_2)$ accepting $L(C)$.
2. Compute the suffix containment relation for M_R (Algorithm S in §3).
3. Construct the intersection graph I of $M_R \times M_C$.
4. Compute the transitive closure I^+ of I.
5. If $[s] \supseteq [t]$ for each edge $((s, i_2), (t, f))$ in I^+, where $f \in F_2$, answer "yes"; otherwise answer "no."

FIG. 7. *Testing whether a regular expression is compatible with a cycle constraint.*

is compatible with $\emptyset$. When the cyclic structure of G is unconstrained, C denotes Σ^+, and every path in T satisfies C, so $[s]$ must contain $[t]$ for all pairs of reachable states in T. This corresponds to the case of restricted regular expressions; that is, a regular expression R is compatible with C (where C denotes Σ^+) if and only if R is restricted. The relationship among these properties is shown in Fig. 6.

By appealing once again to the result of Lemma 3.1, we obtain the following corollary to Theorem 4.1.

COROLLARY 4.2. *Let C be a cycle constraint and G be a db-graph that complies with C. A query Q_R on G can be evaluated in polynomial time in the size of both R and G if R is compatible with C.*

A simple algorithm for testing whether a regular expression is compatible with a cycle constraint is given in Fig. 7. Because it constructs DFAs from regular expressions R and C, the algorithm can take exponential time in the length of R and C. However, deciding whether R and C are compatible is NP-hard, since deciding whether R is restricted is a special case of testing compatibility.

THEOREM 4.3. *Given a regular expression R and a cycle constraint C, deciding whether R and C are compatible is NP-hard.*

Example 7. Let $R = (00)^*$. A DFA M_R accepting $L(R)$ is shown in Fig. 8(a). Because $[a] \not\supseteq [b]$, we know that R is not restricted. In fact, we saw in Theorem 2.1 that deciding if $(x, y) \in Q_R(G)$ is NP-complete for db-graphs in general. However, Q_R can be evaluated in polynomial time on bipartite graphs. As we have already seen, the regular expression $C = (\_\ \_)^+$ defines the class of bipartite graphs. A DFA M_C accepting $L(C)$ is shown in Fig. 8(b), while the intersection graph I of M_R and M_C is given in Fig. 9. The only paths in I satisfying C which start from a node containing the initial state of M_C and end at a node

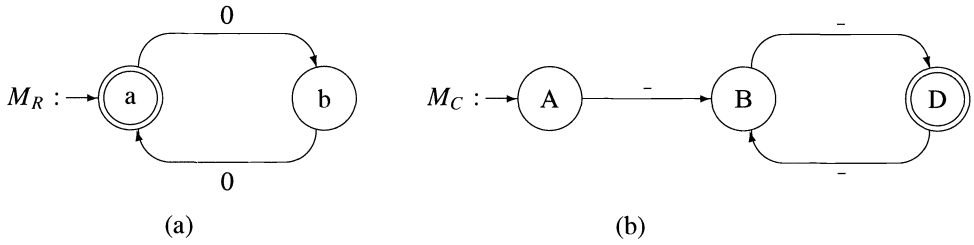


FIG. 8. *DFAs* (a) M_R *for* $R = (00)^*$ *and* (b) M_C *for* $C = (\_\ \_)^+$.

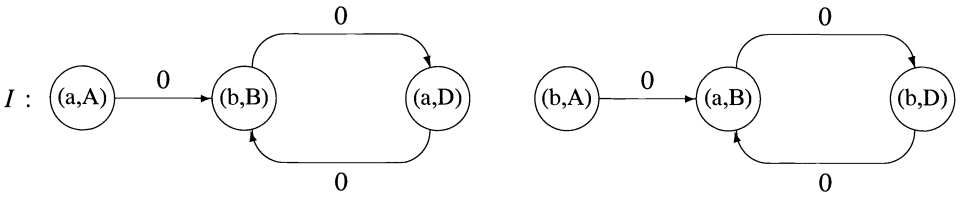


FIG. 9. *The intersection graph* I *of* M_R *and* M_C (*Fig.* 8).

containing a final state of M_C are from (a, A) to (a, D) and from (b, A) to (b, D). Since $[a] \supseteq [a]$ and $[b] \supseteq [b]$, Corollary 4.2 tells us that Q_R can be evaluated in polynomial time on any bipartite graph.

Given a query Q_R and a db-graph G, if we know that G complies with cycle constraint C, we can test whether R is compatible with C using the above algorithm. If so, we can use a polynomial-time algorithm to evaluate Q_R on G. On the other hand, if we do not know about the cyclic structure of G, it seems that we might have to resort to an exponential-time algorithm if R is not restricted. In the next section, however, we describe an evaluation algorithm which runs in polynomial-time in the size of G if G happens to comply with a cyclic constraint with which R is compatible.

5. An evaluation algorithm. In this section, we describe an algorithm for evaluating a query Q_R on a db-graph G. As is to be expected from the results of §2, the algorithm does not run in polynomial time in general. It does, however, run in polynomial time under the sufficient conditions identified in §3 and §4, namely, when G is acyclic, R is restricted, or G complies with a cycle constraint compatible with R. In fact, we show that the algorithm runs in polynomial time if G and R are conflict-free, a condition implied by those above.

The evaluation algorithm traverses paths in G, using a DFA M accepting $L(R)$ to control the search by marking nodes as they are visited. We must record with which state of M a node is visited, since we must allow a node to be visited with different states (which correspond to distinct nodes in the intersection graph of G and M). In order to avoid visiting a node twice in the same state, we would like to retain the state markings on nodes as long as possible. Unfortunately, the following example shows that, in general, requiring answer nodes to be connected by simple paths in G and retaining state markings can lead to incompleteness in query evaluation.

Example 8. Consider the query Q_R, where $R = 0^*1^+0^*$. An automaton M accepting $L(R)$ and a db-graph G are shown in Fig. 10. Note the similarity between M and the automaton of Fig. 3 in §3. Assume that we start traversal from node A in G, and follow the path to B, C, and D. Nodes A, B, C, and D are marked with states a, a, b, and b, respectively, and the answers (A, C) and (A, D) are found, since b is a final state. We cannot mark C with state c

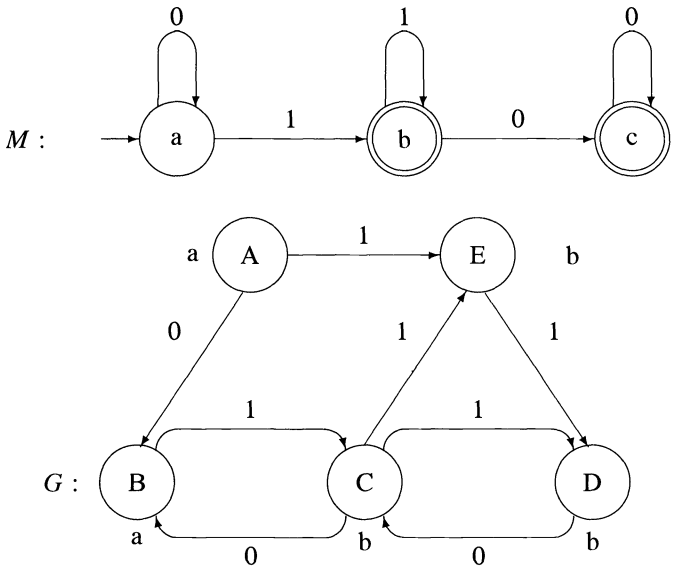


FIG. 10. *A DFA M and db-graph G.*

because (A, B, C, D, C) is a nonsimple path. If we now backtrack to node C, we can mark E with b, resulting in the answer (A, E) being found. Node D is still marked with b (as shown in Fig. 10), so we backtrack to C. However, once again we cannot mark B with state c because (A, B, C, B) is a nonsimple path. So we backtrack to A, and find that E is already marked with state b. Consequently, the search terminates without the answer (A, B) being found.

It turns out that it is safe to retain markings when G is acyclic or R is restricted. However, because of the structure of a particular db-graph G, it might be the case that we can retain markings and evaluate Q_R in polynomial time even if G is not acyclic and R is not restricted.

DEFINITION 15. Let I be the intersection graph of a db-graph G and a DFA $M = (S, \Sigma, \delta, s_0, F)$. An *initial path* in I is any path of the form $((v_0, s_0), \ldots, (v_n, s_n))$. The initial path p is *conflict-free* if (1) p is db-simple, or (2) p is $q \cdot (v, s)$, where q is conflict-free and if v appears in q, then for some (v, t) in q, $[t] \supseteq [s]$. If for no (v, t) in q is it the case that $[t] \supseteq [s]$, then there is a *conflict* at v.

If every simple initial path in I is conflict-free, then I is said to be *conflict-free*,[3] as are G and R.

It is obvious that if G is acyclic, then I is conflict-free no matter what regular expression R appears in Q_R. Also, if R is restricted, then, by Theorem 3.4, M exhibits the containment property; hence, I is conflict-free irrespective of the structure of G. Finally, if G complies with a cycle constraint compatible with R, then, by Theorem 4.1, G and R are conflict-free. We will show that Q_R can be evaluated in polynomial time if I is conflict-free. Hence, conflict-freedom is another (weaker) sufficient condition for Q_R to be polynomial-time evaluable (see Fig. 11).

The result of the following lemma is used in our evaluation algorithm.

LEMMA 5.1. *Let I be the intersection graph of a db-graph G and a DFA $M = (S, \Sigma, \delta, s_0, F)$ accepting $L(R)$. An initial path p in I is conflict-free if and only if* (1) *p is db-simple or* (2) *p is $q \cdot (v, s)$, where q is conflict-free and if v appears in q, then for the* first *(v, t) in q, $[t] \supseteq [s]$.*

[3] This is a strictly weaker definition of conflict-freedom than that given in [18].

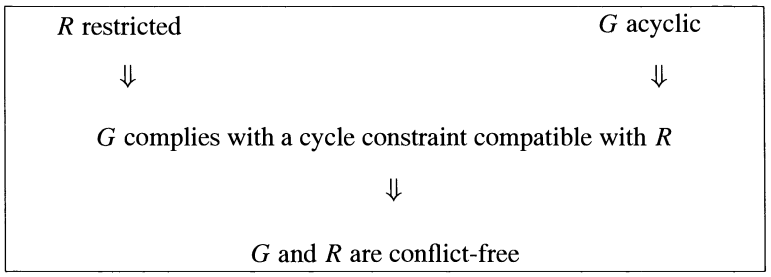


FIG. 11. *Relationship between regular expression R and db-graph G for query* Q_R.

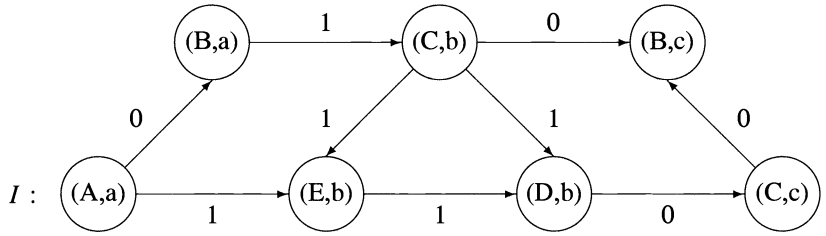


FIG. 12. *The intersection graph I of db-graph G and DFA M of Fig.* 10.

Proof. The "if" direction is trivial. Assume that p is conflict-free but not db-simple. Furthermore, assume that p is $q \cdot (v, s)$, where q is conflict-free and v appears in q. We prove, by induction on the number of occurrences of v in q, that $[t] \supseteq [s]$ where (v, t) is the first occurrence of v in q.

The basis in which v occurs only once in q is trivial. Assume that the inductive hypothesis is true for fewer than n occurrences of v in q, and let p be $q \cdot (v, s)$. Since p is conflict-free, we know from the definition that for some (v, r) in q, $[r] \supseteq [s]$. By the inductive hypothesis, $[t] \supseteq [r]$; hence, $[t] \supseteq [s]$, as required. □

Example 9. Consider again the DFA M and the db-graph G of Example 8 shown in Fig. 10. The intersection graph I of G and M is shown in Fig. 12. Recall that, if markings were retained, the answer (A, B) would not be found. However, there is a conflict in I. This is because there is an initial path in I from (A, a) via (B, a) to (B, c), but $[a] \not\supseteq [c]$. Algorithm C detects such conflicts and unmarks nodes on backtracking, enabling the answer (A, B) to be found.

We now proceed with a description of Algorithm C, shown in Fig. 13. The algorithm uses a DFA $M = (S, \Sigma, \delta, s_0, F)$ accepting $L(R)$ to control a depth-first search of the db-graph G (line 1). There are two reasons why a DFA rather than an NDFA is used. The first is to ensure that no conflicts are encountered when R is restricted. The second reason is to avoid detecting unnecessary conflicts in I. In an NDFA, if $[s] \not\supseteq [t]$, it might be the case that there is a state q such that both s and q are in $\delta^*(s_0, w)$, for some $w \in \Sigma^*$, and $[q] \supseteq [t]$. If node v in G is first marked with s, following which a cycle at v satisfying L_{st} is traversed, a conflict would be registered. This is unnecessary since v would subsequently be marked with q, and any simple path from v satisfying $[t]$ would be found because $[q] \supseteq [t]$.

Algorithm C traverses the transition graph of M and the db-graph G simultaneously, in effect performing a depth-first search of the intersection graph I of G and M. We will often refer to trees of the depth-first search forest generated by Algorithm C. Because of line 5(a), each tree T in the forest is rooted at an initial node of I. When a final node of I is reached,

Algorithm C: Evaluation of a query on a db-graph.

INPUT:
Db-graph $G = (N, E, \psi, \Sigma, \lambda)$, query Q_R.

OUTPUT:
$Q_R(G)$, the value of Q_R on G.

METHOD:

1. Construct an DFA $M = (S, \Sigma, \delta, s_0, F)$ accepting $L(R)$.
2. Initialize $Q_R(G)$ to $\emptyset$.
3. For each node $v \in N$, set $CM[v]$ and $PM[v]$ to $\emptyset$.
4. Test $[s] \supseteq [t]$ for each pair of states s and t in M.
5. For each node $v \in N$,
   (a) call SEARCH(v, v, s_0,*conflict*) (see Fig. 14)
   (b) reset $PM[w]$ to $\emptyset$ for any marked node $w \in N$.

FIG. 13. *Evaluation of a query on a db-graph.*

```
    procedure SEARCH (u, v, s, var conflict)
    /*
        u and v are nodes in the db-graph
        s is a state in the DFA
        conflict is a Boolean flag
    */
6.      conflict ← false
7.      CM[v] ← CM[v] ∪ {s}
8.      if s ∈ F then Q_R(G) ← Q_R(G) ∪ {(u, v)} fi
9.      for each edge in G from v to w with label a do
10.         if δ(s, a) = t and t ∉ CM[w] and t ∉ PM[w] then
11.             if FIRST(CM[w]) = q and ([q] ⊉ [t]) then
12.                 conflict ← true
                else /* CM[w] = ∅ or [q] ⊇ [t] */
13.                 SEARCH (u, w, t, new-conflict)
14.                 conflict ← conflict or new-conflict
                fi
            fi
        od
15.     CM[v] ← CM[v] − {s}
16.     if not conflict then PM[v] ← PM[v] ∪ {s} fi
    end SEARCH
```

FIG. 14. *Search procedure for query evaluation.*

line 8 adds the appropriate pair of nodes from G to $Q_R(G)$. Lines 9 and 10 force the algorithm to consider only paths in G which satisfy R, that is, paths in I.

While the traversal of I is restricted to simple paths, it is not necessarily restricted to db-simple paths; we will prove below that it is safe to traverse non-db-simple paths in the absence of conflicts. Nodes in G are marked with states of M when they are visited. Two sets

of markings are used for each node v: (1) a set of current markings ($CM[v]$) which indicates the states with which v is associated on the current path on the stack of procedure SEARCH (lines 7 and 15), and (2) a set of previous markings ($PM[v]$) which represents earlier markings of v, excluding the current path (line 16). Current markings are used to avoid cycles in I and to detect conflicts, while previous markings are used where possible to prevent a node in G from being visited more than once in the same state during a single execution of line 5(a). The function FIRST applied to marking set $CM[v]$ returns the first state marking for v on the current path, or false if there is no marking.

A node w is visited in state t only if t is not in the previous markings of w and either w is currently unmarked ($CM[w]$ is empty) or the first state marking q for v on the current path is such that $[q] \supseteq [t]$, that is, there is no conflict between q and t at v (lines 10 to 13). Note that there may in fact be a conflict between t and some later marking of v on the current path, but this does not affect the correctness of the algorithm, as we will demonstrate below.

Lines 6, 11, and 12 implement the conflict detection; that is, *conflict* is true if there is a conflict between states q and t at node w. If *conflict* is set to true at line 12, then lines 14, 15, and 16 ensure that the marking of any node which was on the stack at the time the conflict was detected is removed once that node is unstacked. If no conflict occurs on any path rooted at (v, s), then s is added to the previous markings of v in line 16.

In the proofs that follow, we will often say that (v, s), for example, is on the stack of procedure SEARCH. The variables v and s refer to the middle two parameters of SEARCH and correspond to the node (v, s) in the corresponding intersection graph. The reason for excluding the other two parameters of SEARCH is that u (the first) remains unchanged during an execution of Line 5(a), while we are not always concerned about the value of *conflict*. We will also sometimes exclude *conflict* when referring to a particular invocation of SEARCH, for example, SEARCH(u, v, s). Before proving the correctness of Algorithm C, we demonstrate its behaviour by means of an example.

Example 10. Consider again the intersection graph I of Fig. 12. Two possible depth-first search trees (DFSTs) traversed by Algorithm C are shown in Fig. 15. Note that nodes in a DFST can be repeated because of unmarking; for example, node (D, b) appears three times in Fig. 15(a). Dotted edges in the figure lead to nodes for which SEARCH is not called, either because of a conflict (those in (a)), or because the node is already marked via either CM or PM (as in (b)). These latter edges correspond to forward, back, and cross edges in a conventional DFST [2].

Assume that Algorithm C starts traversal from node (A, a), that is, SEARCH(A, A, a) is called at line 5(a), and that the order of traversal is according to the DFST in Fig. 15(a). Since initially nodes B, C, and D have no current marking, line 11 evaluates to false and SEARCH is called successively with (B, a), (C, b), and (D, b). Because b is a final state, (A, C) and (A, D) are added to $Q_R(G)$ by line 8. Although C already has a current marking (namely b), the fact that $[b] \supseteq [c]$ means that line 11 again evaluates to false and SEARCH is called with (C, c). Now because the first marking for B is a and $[a] \not\supseteq [c]$, a conflict is registered at line 12. The algorithm now backtracks, removing current markings (line 15) and not assigning previous markings (line 16).

Considering (B, c) from (C, b) again gives rise to a conflict, so the algorithm tries the path via (E, b). Note that (D, b) and (C, c) are no longer marked so they are revisited, once again giving rise to a conflict. By the time the algorithm backtracks to (A, a) all nodes (other than A) are unmarked, so that the db-simple path to (B, c) can finally be found and (A, B) added to $Q_R(G)$.

If the path to (B, c) via (E, b) had been chosen first by Algorithm C (as in Fig. 15(b)), then no conflicts would have been detected, resulting in previous markings being kept for B,

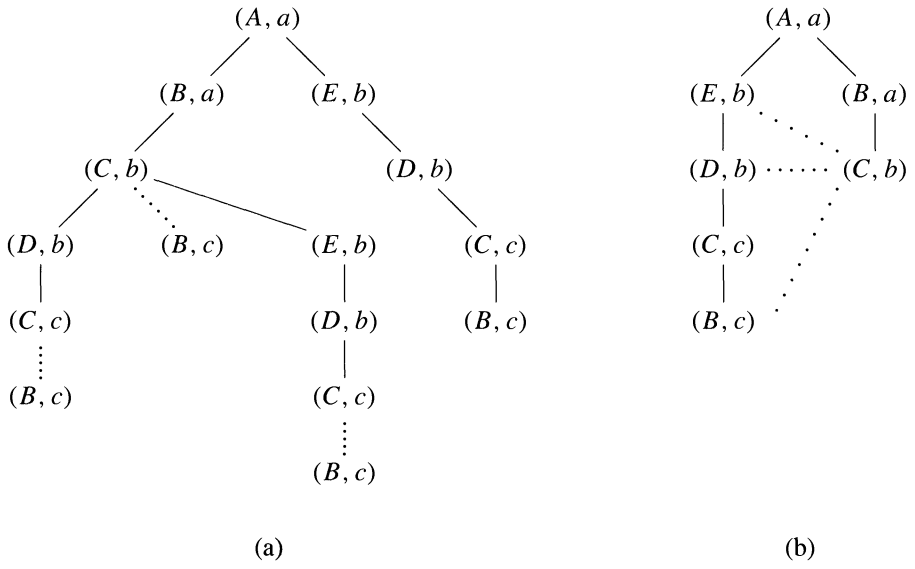


FIG. 15. *Two possible DFSTs.*

D, and E. On traversing the path to (C, b), line 10 would ensure that (B, c), (D, b), and (E, b) are not revisited and no conflicts are registered.

LEMMA 5.2. *If* conflict *is false at line* 16 *of SEARCH(u, v, s), then Algorithm C has performed an entire depth-first search of I from node (v, s).*

Proof. The proof proceeds by induction on the length of the longest simple path p from (v, s) in I. If p is of length zero, the result follows trivially. Assume the result holds for nodes in I from which the longest simple path is of length $n - 1$, and consider node (v, s) for which the longest simple path in I is of length n.

For *conflict* to be false at line 16 of SEARCH(u, v, s), it must be that, for each successor (w, t) of (v, s) in I, either (1) $t \in PM[w]$ at line 10, or (2) *new-conflict* must have been false at line 14. In case (1), *conflict* must have been false at line 16 of SEARCH(u, w, t) in order for t to be added to $PM[w]$. In case (2), *conflict* must have been false at line 16 of SEARCH(u, w, t) so that *new-conflict* is false at line 14. Since the longest simple path from (w, t) in I must be of length less than or equal to $n - 1$, we conclude from the inductive hypothesis that an entire depth-first search from (w, t) has been performed by Algorithm C. Clearly, lines 9 and 10 consider every successor of (v, s) in I, so the result follows. □

DEFINITION 16. A node (v, s) in DFST T is called a *conflict predecessor* if, for some successor (w, t) of (v, s) in I, w appears in an ancestor of (v, s) in T and, for the first such occurrence (from the root), say (w, q), it is the case that $[q] \not\supseteq [t]$. In other words, there is a conflict between q and t at w.

LEMMA 5.3. *Consider the execution of SEARCH(u, v, s) in DFST T. State s is added to $PM[v]$ in line* 16 *if and only if no descendant of (v, s) in T is a conflict predecessor.*

Proof. If s is added to $PM[v]$ in line 16, then *conflict* must be false. Hence, by Lemma 5.2, an entire depth-first search of I from (v, s) must have been performed. But a conflict predecessor is a node (w, t) in T which has a successor in I that does not appear as a successor of (w, t) in T. Thus, no conflict predecessor can appear as a descendant of (v, s) in T.

If no descendant of (v, s) in T is a conflict predecessor, then *conflict* is false for all such descendants and hence for (v, s) itself. Thus, s is added to $PM[v]$ in line 16. □

THEOREM 5.4. *Let $G = (N, E, \psi, \Sigma, \lambda)$ be a db-graph, and R be a regular expression over Σ. Let $M = (S, \Sigma, \delta, s_0, F)$ be a DFA accepting $L(R)$, and I be the intersection graph of G and M. Algorithm C is correct; that is, Algorithm C adds (u, z) to $Q_R(G)$ if and only if there is a db-simple path from (u, s_0) to (z, s_f), $s_f \in F$, in I (that is, there is a simple path from u to z in G satisfying R).*

Proof. Algorithm C clearly terminates, since line 10 ensures that only simple paths in I are considered, and no simple path from an initial node is considered more than once.

(Only if) If the algorithm adds (u, z) to $Q_R(G)$, then it must traverse a DFST T rooted at (u, s_0) in which there is a simple path p from (u, s_0) to (z, r), $r \in F$.

Assume that p is not db-simple, and that the db-node v appears more than once on p. Let the first occurrence of v on p be in I-node (v, s) and the last such occurrence be in (v, t). Thus s was the first state added to $CM[v]$, and in order for SEARCH(u, v, t) to have been called in line 13, line 11 must have ensured that $[s] \supseteq [t]$. Hence, there is a path p' from (v, s) to (z, q), $q \in F$, in I such that the sequence of db-nodes on p' is identical to that on the path from (v, t) to (z, r) on p. Since (v, s) and (v, t) are the first and last occurrences, respectively, of v on p, there is a path from (u, s_0) to (z, q), $q \in F$, in I which is db-simple with respect to v.

A simple induction on the number of repeated db-nodes on p shows that there is a db-simple path from (u, s_0) to (z, s_f), $s_f \in F$.

(If) Assume there is a db-simple path p from (u, s_0) to (z, s_f), $s_f \in F$, in I. Obviously, if the algorithm traverses p we are done. Assume that it does not. Let (v, s) be the last node on p that is traversed, and (w, t) be the successor of (v, s) on p. The reason (w, t) is not visited cannot be because of a conflict, since p is db-simple. So it must have been the case that $t \in PM[w]$ at line 10. By Lemmas 5.2 and 5.3, an entire depth-first search of I from (v, s) must have been performed. Since there is a path from (v, s) to (z, s_f) in I, SEARCH(u, z, s_f) must have been called, in which case (u, z) would have been added to $Q_R(G)$ in line 8. □

THEOREM 5.5. *In the absence of conflicts, Algorithm C runs in an amount of time which is bounded by a polynomial in the size of the db-graph.*

Proof. The essential point is that, in the absence of conflicts, Algorithm C performs a normal depth-first search of the intersection graph which is polynomial in the size of the db-graph. A detailed analysis of the time complexity of the algorithm follows.

Let Q_R be a query where R is of length m, and G be a db-graph with n nodes and e edges. Although there can be as many as $O(2^m)$ states in a DFA accepting $L(R)$, this is just a constant in terms of the size of G. Nevertheless, we will assume that M has q states and will include q in our analysis of the time complexity of Algorithm C. Since M has at most $O(q^2)$ transitions, the intersection graph I for G and M has $O(qn)$ nodes and $O(q^2e)$ edges.

Line 1 of Algorithm C can be done in $O(q^2)$ time, while line 2 requires only constant time. Line 3 takes $O(n)$ time and line 4 $O(q^2)$ time. Line 5 is executed n times, and, in any execution, each node in I is visited at most once if I is conflict-free. This is because when (v, s) is stacked, s is added to $CM[v]$ and line 10 ensures that (v, s) cannot be restacked; when (v, s) is unstacked, s is added to $PM[v]$ (line 16) and is not removed from $PM[v]$ until the present execution of line 5(a) has terminated. Once again, line 10 ensures that (v, s) cannot be revisited during the present execution of 5(a).

Only constant time is needed for lines 6, 12, and 14. For each db-node v, $CM[v]$ can be implemented as a stack with access to its bottom element through the function FIRST. Hence, lines 7, 11, and 15 can be performed in constant time, as can line 16 since $\{s\}$ and $PM[v]$ are disjoint (by line 10). Line 8 can be implemented to take $O(q)$ time: the pair (u, v) is added to $Q_R(G)$ if and only if there is no other final state in $PM[v]$. Line 10 can also be done in $O(q)$ time. Lines 9 and 10 inspect each edge leaving a node in I, and since no node

in I can be revisited, SEARCH can be called $O(q^2e)$ times. Each call takes $O(q)$ time, so a single execution of line 5(a) takes $O(q^3e)$ time. A single execution of line 5(b) takes $O(n)$ time, so the total time spent in line 5 is $O(n(q^3e + n))$. Consequently, Algorithm C runs in $O(n(q^3e + n))$ time. In terms of the size of G, Algorithm C runs in $O(ne)$ time (under the assumption that there are more edges than isolated nodes). □

From the relationship depicted in Fig. 11, we obtain the following.

COROLLARY 5.6. *Algorithm C evaluates Q_R on G in time polynomial in the size of G if*

1. *R is restricted,*
2. *G is acyclic, or*
3. *G complies with a cycle constraint compatible with R.*

Even in the presence of conflicts, Algorithm C can run in polynomial time in the size of G. This is the case, for example, if R is a (*)-free regular expression. Let q be the length of R. If R is (*)-free, there are only a finite number of strings in $L(R)$ and the length of the longest such string is q. This then is also an upper bound on the length of the longest db-simple path in I. Hence, there can be at most $O(n^q)$ db-simple paths in I. So even if Algorithm C traverses every db-simple path in I exactly once (the worst case), it still runs in polynomial time in the size of G.

A number of circumstances other than those identified above can lead to polynomial-time solutions. For example, there are certainly queries that can be evaluated in polynomial time on arbitrary db-graphs but whose regular expressions are not restricted. One such class of regular expressions are those of the form wa^*, where w is a string of fixed length. Unfortunately, there are db-graphs on which Algorithm C takes exponential time to evaluate the associated queries.

Clearly, there is much scope for further investigation. Additional classes of queries/db-graphs for which polynomial-time evaluation is possible should be identified and appropriate, more general evaluation algorithms developed. Algorithm C itself could be enhanced so that it reacts in a more sophisticated manner on detecting a conflict. One possibility is to flag the source of the conflict and not to unmark nodes until the algorithm backtracks from the flagged node.

6. Conclusions. We have addressed the problem of finding nodes in a labelled, directed graph which are connected by a simple path satisfying a given regular expression. This study was motivated by the observation that many recursive queries on relational databases can be expressed in this form, and by the implementation of a query language based on this observation.

We began by describing how a naive algorithm might evaluate such queries. Although this algorithm runs in exponential time in the worst case, we showed that we cannot expect to do better since the evaluation problem is in general NP-hard. Using the fact that the associated problem for paths in general (as opposed to simple paths) is solvable in polynomial time, we characterized the class of restricted regular expressions, whose associated queries can be evaluated in polynomial time.

Having considered restrictions on the structure of regular expressions, we turned our attention to the cyclic structure of the graphs being queried. We introduced the notion of a cycle constraint, and showed that if a graph G complied with a cycle constraint which was compatible with a regular expression R, then $Q_R(G)$ could be evaluated in polynomial time. Finally, we presented an algorithm for evaluating arbitrary expressions on arbitrary graphs. This algorithm runs in polynomial time if (a) the regular expression is restricted or closure-free, (b) the graph complies with a cycle constraint compatible with the regular expression (a special case being when the graph is acyclic), or (c) the regular expression and graph are conflict-free.

While it is difficult to say how often the above conditions will be encountered in practice, we did show that the class of restricted regular expressions is closed under the regular operators. A good starting point for investigation into larger classes of expressions and graphs with polynomial-time evaluation algorithms would be to attempt to identify the class of expressions and graphs which are not conflict-free, but on which Algorithm C runs in polynomial time.

Our emphasis in this paper has been on identifying circumstances in which the regular simple path problem can be solved in polynomial time, rather than designing the most efficient algorithm for these cases. We believe this is a topic for future research. For example, it would be interesting to see whether techniques used on sparse graphs, such as those in [16], could be employed in our algorithm in order to improve its efficiency on sparse graphs.

We should point out that the analysis in this paper, and the implementation itself, assume the graph can be entirely stored in main memory. This is a reasonable assumption in many cases, especially because in the intended applications of our query language $\mathbf{G}^+$ the graph is often only the fraction of the database that can be presented visually in a natural way. Relaxing this assumption provides an interesting area for further study. Other researchers, investigating similar algorithms for transitive closure, have claimed that they are amenable to efficient secondary storage implementation [15].

Finally, we note that research has been done on the expressive power of graph-based query languages in which the restriction of simple path semantics is dropped. One such language that captures exactly the queries computable in nondeterministic logarithmic space is presented in [8]. On-line algorithms for regular path finding are given in [5], while a survey of many results can be found in [24].

REFERENCES

[1] R. AGRAWAL, *Alpha: An extension of relational algebra to express a class of recursive queries*, in Proceedings of the 3rd International Conference on Data Engineering, New York, IEEE, Piscataway, NJ, 1987, pp. 580-590.

[2] A. AHO, J. HOPCROFT, AND J. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[3] A. AHO AND J. ULLMAN, *Universality of data retrieval languages*, in Proceedings of the 6th ACM Symposium on Principles of Programming Languages, New York, Association for Computing Machinery, New York, 1979, pp. 110-120.

[4] F. BANCILHON, *On the completeness of query languages for relational databases*, in Proceedings of the 7th Symposium on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science 64, Springer-Verlag, New York, 1978, pp. 112-123.

[5] A. BUCHSBAUM, P. KANELLAKIS, AND J. VITTER, *A data structure for arc insertion and regular path finding*, in Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms, 1990, pp. 22-31.

[6] B. CARRÉ, *Graphs and Networks*, Oxford University Press, Oxford, England, 1979.

[7] E. CODD, *Relational completeness of data base sublanguages*, in Data Base Systems, R. Rustin, ed., Prentice-Hall, Englewood Cliffs, NJ, 1972, pp. 65-98.

[8] M. CONSENS AND A. MENDELZON, *Graphlog: A visual formalism for real life recursion*, in Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Nashville, TN, Apr. 2-4, 1990, Association for Computing Machinery, New York, pp. 404-416.

[9] I. CRUZ, A. MENDELZON, AND P. WOOD, *A graphical query language supporting recursion*, in Proceedings of the ACM SIGMOD Conference on Management of Data, San Francisco, CA, May 27-29, 1987, Association for Computing Machinery, New York, pp. 323-330.

[10] ———, $\mathbf{G}^+$*: Recursive queries without recursion*, in Proceedings of the 2nd International Conference on Expert Database Systems, Tysons Corner, VA, Apr. 25-27, 1988, Benjamin/Cummings, Redwood City, CA, pp. 355-368.

[11] S. FORTUNE, J. HOPCROFT, AND J. WYLLIE, *The directed subgraph homeomorphism problem*, Theoret. Comput. Sci., 10 (1980), pp. 111-121.

[12] G. GRAHNE, S. SIPPU, AND E. SOISALON-SOININEN, *Efficient evaluation for a subset of recursive queries*, in Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, San Diego, CA, Mar. 23-25, 1987, Association for Computing Machinery, New York, pp. 284-293.

[13] J. HOPCROFT AND J. ULLMAN, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[14] H. HUNT, D. ROSENKRANTZ, AND T. SZYMANSKI, *On the equivalence, containment, and covering problems for the regular and context-free languages*, J. Comput. System Sci., 12 (1976), pp. 222-268.

[15] Y. IOANNIDIS AND R. RAMAKRISHNAN, *Efficient transitive closure algorithms*, in Proceedings of the 14th International Conference on Very Large Data Bases, Los Angeles, CA, Aug. 29-Sept. 1, 1988, Morgan Kaufmann, Palo Alto, pp. 382-394.

[16] B. JOUMARD AND M. MINOUX, *An efficient algorithm for the transitive closure and a linear worst-case complexity result for a class of sparse graphs*, Inform. Process. Lett., 22 (1986), pp. 163-169.

[17] A. LAPAUGH AND C. PAPADIMITRIOU, *The even-path problem for graphs and digraphs*, Networks, 14 (1984), pp. 507-513.

[18] A. MENDELZON AND P. WOOD, *Finding regular simple paths in graph databases*, in Proceedings of the 15th International Conference on Very Large Data Bases, Amsterdam, The Netherlands, Aug. 22-25, 1989, Morgan Kaufmann, Palo Alto, CA, pp. 185-193.

[19] A. ROSENTHAL, S. HEILER, U. DAYAL, AND F. MANOLA, *Traversal recursion: A practical approach to supporting recursive applications*, in Proceedings of the ACM SIGMOD Conference on Management of Data, Washington, DC, May 28-30, 1986, Association for Computing Machinery, New York, pp. 166-176.

[20] Y. SHILOACH, *A polynomial solution to the undirected two paths problem*, J. Assoc. Comput. Mach., 27 (1980), pp. 445-456.

[21] L. STOCKMEYER AND A. MEYER, *Word problems requiring exponential time*, in Proceedings of the 5th Annual ACM Symposium on Theory of Computing, Austin, TX, Apr. 30-May 2, 1973, Association for Computing Machinery, New York, pp. 1-9.

[22] R. TARJAN, *Fast algorithms for solving path problems*, J. Assoc. Comput. Mach., 28 (1981), pp. 594-614.

[23] J. ULLMAN, *Implementation of logical query languages for databases*, ACM Trans. Database Systems, 10 (1985), pp. 289-321.

[24] M. YANNAKAKIS, *Graph-theoretic methods in database theory*, in Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Nashville, TN, Apr. 2-4, 1990, Association for Computing Machinery, New York, pp. 230-242.

SIAM J. COMPUT.
Vol. 24, No. 6, pp. 1259-1277, December 1995

008

WHAT CAN BE COMPUTED LOCALLY?*

MONI NAOR† AND LARRY STOCKMEYER‡

Abstract. The purpose of this paper is a study of computation that can be done locally in a distributed network, where "locally" means within time (or distance) independent of the size of the network. *Locally checkable labeling* (*LCL*) problems are considered, where the legality of a labeling can be checked locally (e.g., coloring). The results include the following:

- There are nontrivial LCL problems that have local algorithms.
- There is a variant of the dining philosophers problem that can be solved locally.
- Randomization cannot make an LCL problem local; i.e., if a problem has a local randomized algorithm then it has a local deterministic algorithm.
- It is undecidable, in general, whether a given LCL has a local algorithm.
- However, it is decidable whether a given LCL has an algorithm that operates in a given time t.
- Any LCL problem that has a local algorithm has one that is order-invariant (the algorithm depends only on the order of the processor IDs).

Key words. distributed computation, local computation, graph labeling problem, resource allocation, dining philosophers problem, randomized algorithms

AMS subject classifications. 68M10, 68Q20, 68Q22, 68R05, 68R10

1. Introduction.

A property of distributed computational systems is *locality*. Each processor is directly connected to at most some fixed number of others. Despite the locality of connections, we may want to perform some computation such that the values computed at different nodes must fit together in some global way. The purpose of this paper is to attempt to understand what can be computed when algorithms must satisfy a strong requirement of locality, namely, that the algorithm must run in *constant time* independent of the size of the network. A processor running in constant time t must base its output solely on the information it can collect from processors located within radius t from it in the network. Apart from the obvious advantage of constant time (that constant time takes less time than nonconstant time), another advantage is improved fault-tolerance: if the algorithm runs in constant time, a failure at a processor p can only affect processors in some bounded region around p. Another motivation for locality is in recent work on self-stabilizing distributed algorithms; for example, Afek, Kutten, and Yung [2] introduced the idea of detecting an illegal global configuration by checking local conditions.

Our work has three goals: first, to lay some groundwork for studying the question of what can and cannot be computed locally; second, to establish some basic, general results; third, to study particular examples.

A network is modeled as an undirected graph, where each node represents a processor and edges represent direct connections between processors. We consider only networks of bounded degree. Our main focus is on computational problems of producing "labelings" of the network. Since our subject is constant time algorithms, it makes sense to restrict ourselves to labelings such that the validity of a labeling can be checked locally (i.e., by checking within

*Received by the editors August 27, 1993; accepted for publication (in revised form) June 28, 1994. A preliminary version of this paper appeared in the Proceedings of the 25th ACM Symposium on Theory of Computing, 1993, pp. 184-193.

†Department of Applied Mathematics and Computer Science, Weizmann Institute of Science, Rehovot 76100, Israel (naor@wisdom.weizmann.ac.il). The research of this author was partly supported by a grant from the Israel Science Foundation administered by the Israeli Academy of Sciences. This work was performed while the author was at the IBM Almaden Research Center.

‡IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (stock@almaden.ibm.com).

some fixed radius from the node). We call these *locally checkable labelings* (*LCLs*). Familiar examples of LCLs are vertex coloring, edge coloring, and maximal independent set (MIS). In the case of MIS, for example, one local constraint says that if vertex v is in the MIS then no neighbor of v is in the MIS; another constraint says that if v is not in the MIS then v has at least one neighbor in the MIS. In general, the output labeling might depend on some initial input labeling, and most of our general results hold in this case. If all processors are identical, it is already known (by familiar symmetry arguments) that the types of labeling problems that can be solved deterministically are very limited. So we assume that processors are given unique numerical IDs. If an algorithm runs in time t then, for each vertex v, the processor at v can collect information about the structure of the network, including processor IDs (and possibly input labels), in the region of radius t around v. Then the processor must choose its output label based on this information. The algorithm must be correct, that is, the entire output labeling must be valid, regardless of how the processors are numbered with unique IDs.

Several recent papers have given improved time algorithms for certain LCLs such as MIS and vertex coloring, for example, Awerbuch et al. [3], Goldberg, Plotkin, and Shannon [8], Linial [10], and Panconesi and Srinivasan [15]. However, these papers do not consider constant time; the running time of the algorithms grows with the size of the network. Indeed the time must grow. In the first paper to establish the limitations of locality in this context, Linial [10] proved that, even on ring networks, an MIS or a 3-coloring of vertices cannot be found in constant time.[1]

In light of previous work on locality, two questions come to mind:

- Can any nontrivial LCL problem be solved in constant time?
- If the answer to the first question is "yes," can we characterize the LCL's that can be solved in constant time?

One of our results is that the answer to the first question is "yes." Define a *weak c-coloring* of a graph to be a coloring of the vertices with c colors such that each nonisolated vertex has at least one neighbor colored differently. It is easy to see that a weak 2-coloring exists for every graph. We show the following for every fixed d:

- Consider the class of graphs of maximum degree d, where every vertex has odd degree. There is a $c = c(d)$ and an algorithm that finds a weak c-coloring in time 2 for any graph in this class. Here c is exponential in d, but in an additional time $O(\log^* d)$ the number of colors can be reduced to 2.

This result is the best possible in three senses:

- For d-regular graphs where d is even, for no constant $c = c(d)$ is there a constant time algorithm that finds a weak c-coloring.
- The time bound 2 cannot be reduced to 1.
- If we change the definition of a coloring so that every vertex v must have at least two neighbors colored differently than v then, even for d-regular graphs with d odd, a coloring cannot be found in constant time.

Although a weak coloring might seem a strange concept, we have used it as a basis for a solution to a certain resource allocation problem. A well-known paradigm for resource allocation problems is Dijkstra's dining philosophers problem, which was later generalized from a ring to arbitrary graphs (see, e.g., [4], [11]). In the version of the problem we consider, there is a given *conflict graph* where each node represents a processor and each edge represents a resource (a "fork") which is shared by the two endpoint processors. It is assumed that if two processors share a resource, then they are also close in the communication network. At any time a fork can be "owned" by at most one of the processors that share it. Each processor can

[1] Actually, Linial gives a lower bound of $\Omega(\log^* n)$ on oriented rings of size n, which matches an upper bound of Cole and Vishkin [6] to within a constant factor.

be in one of three states: resting, hungry, or eating. The processors operate asynchronously. A resting processor can become hungry at any time. In order to eat, a processor must obtain certain forks; we get different types of problems depending on precisely what "certain" means. A processor eats for at most a bounded time, after which it returns to the resting state. A processor p can attempt to "grab" a certain fork, and can release an owned fork. The grab operation will fail if the fork is currently owned by the other processor q; if this occurs, p may decide to wait for q to release the fork. We require a solution that is starvation free, meaning that a hungry processor will eventually be able to eat. An important measure of the goodness of a solution is the maximum length of a *waiting chain* that can develop. As pointed out by Choy and Singh [5], a difficulty with long waiting chains is that if a processor p fails while holding a fork, the failure will affect every processor behind p in the waiting chain.

In the traditional version of this problem, if a processor shares d forks (has d incident edges in the conflict graph), it can eat only when it has obtained all d forks. In this case, Lynch [11] gave a solution with waiting chains of length $O(c)$, assuming that the conflict graph is edge colored with c colors. The maximum length was reduced to $O(\log c)$ by Styer and Peterson [17], again assuming that an edge coloring is given. Choy and Singh [5] give a solution with waiting chains of length at most 3, assuming that a certain vertex coloring with $d+1$ colors is given. All of these solutions require that the conflict graph be initially colored in some way. Such colorings (provably) cannot be found in constant time. It is therefore natural to ask whether there is any *purely local* solution to this problem, i.e., a solution with waiting chains bounded by a constant, and which does not assume any initial coloring of the conflict graph. In fact, it can be shown that there is no local solution to this problem by reducing the MIS problem to it. However, we show that there is a purely local solution to a relaxed version of the problem. In this version, a processor can eat when it has obtained *any two* forks. This can be viewed as a threshold condition: a processor can proceed when it has two units of resource. We call this problem the *formal-dining philosophers* problem. Imagine that dining is formal and, in order to eat, a philosopher must dress formally and in particular wear cuff links. We assume that the resource on each edge is a cuff link. In order to dress formally (in the western male tradition) and eat, the philosopher must get any two cuff links. Our solution works in any bounded degree conflict graph of minimum degree 3, i.e., every vertex has at least 3 incident edges. (If the degree is 2, then we have Dijkstra's original version on a ring, for which it is impossible to find a local solution.) To the best of our knowledge, this is the first nontrivial resource allocation problem that has been solved in a purely local fashion.

Returning to the second question above (Can we characterize the LCLs that can be solved in constant time?), another result shows that this will be difficult—it is undecidable. Fix any $d \geq 3$ and let $\mathcal{G}$ be the class of d-regular graphs or the class of graphs of maximum degree d. Even if we restrict attention to LCLs such that every graph in $\mathcal{G}$ has a legal labeling, we show that it is undecidable, given an LCL $\mathcal{L}$, whether there is a constant time algorithm that solves $\mathcal{L}$ for every graph in $\mathcal{G}$. If $d = 2$, however, the problem becomes decidable. The problem is also decidable if we are given a specific time t and would like to know whether there is a t-time algorithm for the given LCL instance.

We close this introduction by mentioning two additional "general" results. The first states that there is no loss of generality in restricting attention to algorithms that do not use the actual values of the processor IDs, but only their relative order. This result is useful in proving some of our other results. The proof is by a Ramsey theory argument similar to ones in [18], [7], [13]. This is in contrast to the non-constant-time case, where, for instance, an order-invariant algorithm for 3-coloring the ring would take time $\Theta(n)$, but the Cole–Vishkin [6] method (which uses the actual values of the IDs) takes time $O(\log^* n)$.

Another result states that randomization does not help in solving LCLs in constant time. For the class $\mathcal{G}$ of d-regular graphs or the graphs of maximum degree d for any fixed $d \geq 2$, if

there is a randomized algorithm that runs in time t and solves the LCL $\mathcal{L}$ with error probability $\varepsilon < 1$ on any graph in $\mathcal{G}$, then there is a deterministic algorithm that runs in time t and solves $\mathcal{L}$ on any graph in $\mathcal{G}$.

We now outline the remainder of the paper. Section 2 gives our definitions of LCLs and local algorithms. In §3 we show that every local algorithm can be replaced with an order-invariant one. The subject of §4 is undecidability and decidability of questions about local solvability. In §5 we show that randomization does not help in solving LCLs locally. The subject of §6 is weak coloring. In §7, the local algorithm for weak coloring is used, together with other ideas, to give a local solution to the formal-dining philosophers problem. In §8, we suggest some open questions raised by our work. For readers interested mainly in the results for weak coloring and formal-dining philosophers, we should point out that §§6 and 7 are completely independent from §§4 and 5. In addition, the local algorithms for weak coloring and formal-dining philosophers do not depend on anything from §§3, 4, or 5, although the impossibility results for weak coloring and formal-dining philosophers use the order-invariance result from §3.

2. Definitions. We first give some definitions and notation concerning graphs. All graphs in this paper are simple and undirected. For a graph $G = (V, E)$ and vertices $u, v \in V$, let $dist_G(u, v)$ be the distance (length of a shortest path) in G from u to v. If $u \in V$ and $e \in E$, and if the endpoints of e are v and w, then $dist_G(u, e) = \min\{dist_G(u, v), dist_G(u, w)\} + 1$. For a vertex u and a nonnegative integer r, let $B_G(u, r)$ denote the subgraph of G consisting of all vertices v and edges e such that $dist_G(u, v) \leq r$ and $dist_G(u, e) \leq r$. The subscript G is omitted when G is clear from context. A *centered graph* is a pair (H, s), where H is a graph and s is a vertex of H. The *radius* of (H, s) is the maximum distance from s to any vertex or edge of H.

We now define the notion of a LCL. For simplicity, we give the definition only for vertex labelings. A similar definition can be given for edge labelings (e.g., edge colorings or edge orientations). To make the definition somewhat more general, we allow the vertices of the graph to be initially labeled with "input labels." Formally, then, an LCL $\mathcal{L}$ consists of a positive integer r (called the *radius* of $\mathcal{L}$), a finite set Σ of *input labels*, a finite set Γ of *output labels*, and a finite set $\mathcal{C}$ of *locally consistent labelings*. Each element of $\mathcal{C}$ is a centered graph of radius at most r, where each vertex is labeled with a pair from $\Sigma \times \Gamma$. Given a graph $G = (V, E)$ and a labeling $\lambda : V \to \Sigma \times \Gamma$, the labeling λ is *$\mathcal{L}$-legal* if, for every $u \in V$, there is an $(H, s) \in \mathcal{C}$ and an isomorphism π mapping $B_G(u, r)$ to H such that $\pi(u) = s$ and π respects the labeling; i.e., for every w, the label-pair of w equals the label-pair of $\pi(w)$. Although certain types of labelings, such as the usual definition of vertex coloring, are more naturally expressed in terms of *forbidden conditions* instead of allowed conditions, it is easy to see that the definition above captures such labelings. Essentially, the set $\mathcal{C}$ gives a "truth table" of all locally consistent labelings. Many of our specific examples of LCLs do not have input. Such LCLs are a special case of the definition above simply by taking $|\Sigma| = 1$.

We consider distributed algorithms which operate on graphs G that are initially input-labeled and where each vertex is also numbered with a unique positive integer ID. If the algorithm produces an output label for each vertex within t steps, we can assume that, for each vertex u, the part of the algorithm running at u collects information about the structure, input labels, and IDs of $B_G(u, t)$, and chooses an output label for u based on this information (although particular algorithms might not actually "use" all this information). Suppose that the algorithm is to be run on graphs of maximum degree d. For a constant t, a *local algorithm with time bound t* is a function A; the input to A is a centered graph (H, s) of radius at most t and degree at most d, whose vertices are labeled with (input, ID) pairs; the value of $A((H, s))$ is some $\gamma \in \Gamma$. The local algorithm A is applied to an input-labeled and ID-numbered graph

G by applying A independently at each vertex of G; that is, for each vertex u, the output label of u is $A(B(u,t))$, where $B(u,t)$ is viewed as a centered graph with center u. For a local algorithm A, an LCL $\mathcal{L}$, and a class $\mathcal{G}$ of graphs, we say that *A solves $\mathcal{L}$ for $\mathcal{G}$* if, for every $G \in \mathcal{G}$, every input labeling of G, and every numbering of the vertices of G with unique IDs, A produces an $\mathcal{L}$-legal labeling; i.e., the combination of the output labeling produced by A with the initial input labeling is $\mathcal{L}$-legal.

Since the subject of the paper is locality, we largely restrict attention to (infinite) classes of graphs for which membership in the class can be checked locally. Examples are d-regular graphs and graphs of maximum degree d for any constant d. Note that if membership in $\mathcal{G}$ can be checked locally, then $\mathcal{G}$ is *closed under disjoint union*; i.e., for every $G, G' \in \mathcal{G}$, the graph consisting of the disjoint union of G and G' belongs to $\mathcal{G}$. We consider only classes with some constant upper bound on degree.

Remark. Although it might be more natural to assume that the IDs for an n-vertex graph are drawn from $\{1, 2, \ldots, n\}$, there is no harm in requiring algorithms to handle arbitrary ID numberings. Suppose that A incorrectly labels G when IDs are arbitrary. Form a new graph G' with n' vertices consisting of the disjoint union of G with a large enough graph so that the vertices of G' can be numbered from $\{1, 2, \ldots, n'\}$ while keeping the numbering of G the same. Then A labels G' incorrectly.

3. Order-invariant algorithms. In what follows, it is sometimes useful to restrict attention to algorithms that do not use the actual values of the IDs, but only their relative order. Two ID numberings η and η' of a graph H are *order-equivalent* if, for every pair of vertices u and v, $\eta(u) < \eta(v)$ iff $\eta'(u) < \eta'(v)$. A local algorithm A is *order-invariant* if for every (H, s) in the domain of A, if we obtain H' from H by changing the ID numbering η to any other η' such that η and η' are order-equivalent, then $A((H, s)) = A((H', s))$.

Using Ramsey theory, we show that there is no loss of generality in restricting attention to order-invariant algorithms. This type of application of Ramsey theory is hardly new: starting with Yao's celebrated paper on searching tables [18], through Frederickson and Lynch's paper [7] on a problem in distributed computing, and Moran, Snir, and Manber's [13] work on decision trees, there have been many papers on the subject.

For a set S and an integer $p \leq |S|$, let $[S]^p$ denote the set of subsets $A \subseteq S$ with $|A| = p$. We use the following theorem of Ramsey [16]. (For information on Ramsey theory see [9].)

THEOREM 3.1 (Ramsey). *For any p, m, and c, there is a number $R(p, m, c)$ such that the following holds: Let S be a set of size at least $R(p, m, c)$. For any coloring of $[S]^p$ with at most c colors, there is a $T \subseteq S$ with $|T| = m$ such that all of $[T]^p$ is colored the same.*

We first state and prove the order-invariance result in a stronger form which will be useful later.

LEMMA 3.2. *Fix an LCL $\mathcal{L}$ (with or without input), a class $\mathcal{G}$ of graphs, and a time bound t. Let d be the maximum degree of a graph in $\mathcal{G}$. There is a number R, depending only on d, t, and $\mathcal{L}$, such that the following holds: For every local algorithm A with time bound t and every set S of IDs with $|S| \geq R$, there is an order-invariant local algorithm A' with time bound t such that, for every $G \in \mathcal{G}$ and every input labeling of G, if A labels G correctly for every ID numbering drawn from S then A' labels G correctly for every ID numbering.*

Proof. We show how to convert any A to an order-invariant A' such that the last sentence of Lemma 3.2 is satisfied. Let $(K_1, s_1), \ldots, (K_z, s_z)$ be the set of input-labeled centered graphs (K, s) such that ID numberings of (K, s) appear in the domain of A. Let p be the maximum number of vertices in any K_i. Note that p and z depend only on d, t, and $\mathcal{L}$.

Given any set S of IDs, define an equivalence relation on $[S]^p$ as follows. For $X, X' \in [S]^p$, let

$$X = \{x_1, x_2, \ldots, x_p\} \text{ and } X' = \{x'_1, x'_2, \ldots, x'_p\}$$

be the elements of X and X' indexed in increasing order. Viewing $K_1, \ldots, K_z$ as graphs on disjoint sets of vertices, let V be the union of all these vertex sets. For $\sigma : V \to \{1, 2, \ldots, p\}$, let $K_j(\sigma)$ (resp., $K'_j(\sigma)$) be the graph K_j where each vertex v is numbered with the ID $x_{\sigma(v)}$ (resp., $x'_{\sigma(v)}$). We restrict attention to those σ's such that, for every K_j, no two vertices of K_j are numbered the same. Now $X \equiv X'$ iff $A((K_j(\sigma), s_j)) = A((K'_j(\sigma), s_j))$ for all σ and j. It is easy to see that this is an equivalence relation. It is also clear that there is an upper bound c on the number of equivalence classes. This bound depends only on p, z, and $\mathcal{L}$, so it depends only on d, t, and $\mathcal{L}$. Let r be the radius of $\mathcal{L}$ and let m equal p plus the maximum number of vertices in any centered graph of degree at most d and radius at most $r + t$. Again, m depends only on d, t, and $\mathcal{L}$. Let $R = R(p, m, c)$, so R depends only on d, t and $\mathcal{L}$.

Carrying out the above for any S with $|S| \geq R$, Theorem 3.1 implies that there is a set of IDs $T \subseteq S$ with $|T| = m$ such that all members of $[T]^p$ are equivalent. Let U equal T minus the p largest members of T. By choice of m, $|U|$ is as large as the maximum number of vertices in any centered graph of degree at most d and radius at most $r + t$. We claim that A is order-invariant when IDs are drawn from U. Let (H, s) be any centered graph in the domain of A with ID numbering η mapping its vertices to U. Let (H', s) be this graph with the ID numbering η' mapping to U, where η, η' are order-equivalent. Let X (resp., X') be a member of $[T]^p$ containing all the IDs of H (resp., H') such that, for every vertex u, if $\eta(u) = x_i$ then $\eta'(u) = x'_i$ (where, as above, the elements of X and X' are indexed in increasing order). This is possible because T contains p "extra" elements not belonging to U, so in the case in which H has fewer than p vertices, we can use the extra elements to pad the sets X and X' to be of size exactly p. Let σ and j be such that $(H, s) = (K_j(\sigma), s_j)$. By choice of X and X', $(H', s) = (K'_j(\sigma), s_j)$. So $A((H, s)) = A((H', s))$ since $X \equiv X'$.

The algorithm A' works as follows: On a centered graph (H, s) numbered with IDs, A' first changes the IDs in an order-equivalent way to IDs in U. (Since U is large enough, this is always possible.) Then A' answers the same as A on the newly numbered H. Note that since A is order-invariant on U, it does not matter exactly how A' does the renumbering, provided that it is order-equivalent. Clearly A' is order-invariant.

It remains to show that A' has the required correctness property. Let $G \in \mathcal{G}$ and fix some input labeling. Suppose that A' does not label G correctly. This means that there is some vertex u of G such that $B(u, r)$ is not labeled correctly. Obtain a new ID-numbered graph by changing the IDs to IDs in S in such a way that (i) each vertex of $B(u, r + t)$ has a new ID in U and (ii) the new ID-numbering of $B(u, r + t)$ is order-equivalent with the old one. But since A' is order-invariant and A is order-invariant when IDs are drawn from U, it follows from the definition of A' that, for each vertex v of $B(u, r)$, the output label given to v by A' under the original ID numbering is the same as the output label given to v by A under the new ID numbering. This contradicts the assumption that A correctly labels G when IDs are drawn from S. □

The following theorem is now immediate.

THEOREM 3.3. *Fix an LCL $\mathcal{L}$ and a class $\mathcal{G}$ of graphs. If there is a local algorithm A with time bound t that solves $\mathcal{L}$ for $\mathcal{G}$ then there is an order-invariant local algorithm A' with time bound t that solves $\mathcal{L}$ for $\mathcal{G}$.*

4. Undecidability. In this section we consider the problem, for a fixed class $\mathcal{G}$ of graphs, of deciding whether a given LCL $\mathcal{L}$ can be solved in constant time for $\mathcal{G}$. The answer could be "no" for an uninteresting reason, namely, that there is some $G \in \mathcal{G}$ that has no $\mathcal{L}$-legal labeling. Therefore we restrict attention to $\mathcal{L}$'s for which every $G \in \mathcal{G}$ has an $\mathcal{L}$-legal labeling.

We also restrict attention to LCLs without input; since our main result is an undecidability result, this just makes the result stronger. Define $Y(\mathcal{G})$ (resp., $N(\mathcal{G})$) to be the set of LCLs $\mathcal{L}$ without input such that every $G \in \mathcal{G}$ has an $\mathcal{L}$-legal labeling and there is (resp., is not) a constant t such that some local algorithm with time bound t solves $\mathcal{L}$ for $\mathcal{G}$. Recall that sets Y and N are *recursively separable* if there is a Turing machine that answers "yes" on every input from Y and answers "no" on every input from N (and we do not care about its answer otherwise).

THEOREM 4.1. *Fix any $d \geq 3$ and let $\mathcal{G}$ be the class of d-regular graphs or the class of graphs of maximum degree d. Then $Y(\mathcal{G})$ and $N(\mathcal{G})$ are not recursively separable.*

Proof. We show that if $Y(\mathcal{G})$ and $N(\mathcal{G})$ are recursively separable, then it can be decided for a given Turing machine M whether M halts on blank tape. We first describe the proof for the class of 4-regular graphs. We begin by proving the result for a different class of graphs and then work in several steps toward 4-regular graphs.

(1) Consider first the class of two-dimensional grid graphs where one corner of the graph is marked as "special," say by having an extra edge which connects it to a new vertex of degree 1. (A two-dimensional grid graph has vertices $\{1, \ldots, k\} \times \{1, \ldots, l\}$ for some k and l, and two vertices are connected by an edge if the L_1-distance between them is 1.) Imagine that the special corner is the upper left corner. Let M be a given Turing machine with states Q and tape alphabet T. Modify M if necessary so that (i) if M does not halt then M visits an infinite amount of tape and (ii) the head never moves left of its initial position. The idea is to have the LCL $\mathcal{L}$ force the labeling to be a computation of M started on blank tape, where the ith row of the grid contains the configuration (tape contents, state, and head position) at the ith step. The head position is given by writing the state symbol just to the left of the scanned symbol. Since a computation has two senses of direction, left versus right on the tape and up (past) versus down (future) in the time dimension, the LCL will also force consistent senses of direction on the grid, at least in the part of the grid that contains the computation. We imagine the senses of direction as giving direction to the edges of the grid from left to right and from up to down (i.e., from past to future).

The construction of the labeling problem is not difficult conceptually, since it is well known that the validity of a Turing machine computation can be checked locally. For definiteness, we describe one way of carrying out the details. A vertex label has the form $\langle \sigma, i, j \rangle$, where $\sigma \in Q \cup T \cup \{I\}$, $0 \leq i \leq 2$, and $0 \leq j \leq 1$. σ is called the *s-label* of the vertex (where s stands for "symbol"). Let v and v' be adjacent vertices with labels $\langle \sigma, i, j \rangle$ and $\langle \sigma', i', j' \rangle$. If $j = j'$ then there is a "horizontal" (left-to-right) edge from v to v' iff $i' = i + 1 \bmod 3$. If $j \neq j'$ then there is a "vertical" (up-to-down) edge from v to v' iff $i' = i + 1 \bmod 3$. The s-label I means that the vertex is "inactive," i.e., it is not in the part of the grid that contains the computation.

The LCL $\mathcal{L}$ enforces the following constraints:

1. The s-label of the special corner must be the initial state of M and the two senses of direction must be directed away from this corner.

2. The senses of direction propagate correctly. This can be done, for example, by requiring the senses of direction to be consistent on every 3×3 subgrid. However, we do not require any sense of direction between two adjacent inactive vertices.

3. Each vertex on the upper boundary of the grid, other than the special corner, has s-label either I or the blank tape symbol.

4. In the vicinity of a state symbol, the computation must proceed according to the transition rules of M. However, we allow the state symbol to disappear if the head attempts to move off the right boundary or the bottom boundary of the grid.

5. In a neighborhood that does not contain a state symbol, each row must be identical to the row above it, except that vertices can become inactive. That is, if there are left-to-right edges from v_1 to v_2 and from v_2 to v_3, if there is an up-to-down edge from v_2 to v_4, and if none

of v_1, v_2, v_3 are s-labeled by a state symbol, then the s-label of v_4 must be either the s-label of v_2 or I.

6. There is no up-to-down edge from an inactive vertex to an active vertex (i.e., once a tape cell becomes inactive, it cannot become active at a later time).

7. A nonhalting state symbol cannot be adjacent to an inactive vertex.

Suppose that M halts in t steps when started on blank tape. Assume for the moment that the grid is $k \times l$, where $k, l \geq t + 1$. Then there is a legal labeling where the s-labeling of the upper left $(t + 1) \times (t + 1)$ subgrid describes a halting computation of M on blank tape and the other vertices are inactive. This labeling can be found by a local algorithm with time bound $2t + 3$. This algorithm works as follows at a vertex v. If v lies within the $(t + 2) \times (t + 2)$ subgrid having the special corner as one of its corners, then the position of v in this subgrid is known, and the label of v can be found, since it depends only on this position. Otherwise, v is labeled $\langle I, 0, 0 \rangle$ (recall that the senses of direction do not have to be maintained within an inactive region). The argument for a smaller grid is similar (recall that the head can "move off" the grid at the right and bottom boundaries, so a legal labeling exists).

Suppose now that M does not halt. An argument similar to the one just given shows that a legal labeling exists (in particular, all vertices are active). Assume that a local algorithm A with time bound t finds a legal labeling for any ID-numbering of any grid. Using Theorem 3.3, convert A to an order-invariant algorithm A' with time bound t. For a given grid, consider the ID-numbering where each row is labeled from left to right in increasing order and the IDs used for row i are all smaller than those used for row $i + 1$. Since A' is order invariant, A' will assign the same label to any two vertices v and w that are both farther than distance t from any boundary of the grid, since any two such points look the same to A'. Since M uses an infinite amount of tape, it is clear that the labeling produced by A' is not legal if the grid is sufficiently large.

(2) We now consider two-dimensional grid graphs where no corner is "special," so all four corners look identical locally. A problem with the previous construction is that now there can be four computations, one starting at each corner. If M does not halt, or if the grid is too small, the senses of direction of these computations will conflict. The problem is solved by having four "levels." Now a label has the form $\langle \lambda_1, \lambda_2, \lambda_3, \lambda_4 \rangle$, where each λ_i is a label as in part (1). The constraints that must hold at a special corner in part (1) now must hold at each corner, but only on one level. The arguments that a legal labeling always exists and M halts iff a legal labeling can be found in constant time are essentially identical to those of part (1). In the case in which M halts and the grid is so small that 2–4 computations overlap, the constant time algorithm uses the order of the IDs at the relevant corners to decide which computation to put on which level.

(3) The next step is to consider a class of 4-regular graphs. These are grid graphs with extra edges added around the boundary to make the graph 4-regular. This can be done in such a way that the boundary vertices and the corner vertices can be identified locally. Call these graphs 4-*regular grids*.

(4) Finally we consider the entire class of 4-regular graphs. Call a vertex a *defect* if it does not look locally like a 4-regular grid, i.e., for some suitably large (but constant) c, $B(v, c)$ is not consistent with a 4-regular grid. One problem with the previous construction is that now there can be many vertices that look locally like the corner of a 4-regular grid, so many different computations will be started and might conflict with one another, e.g., turn a halting computation into a nonhalting one. This can occur, however, only if the graph has defects. The new idea is to propagate a chain of "erasing symbols" E from the defect back to the corner, so that the computation does not have to start.

More precisely, the s-symbols now include E also. For two adjacent vertices with s-symbol E, we use the component i of a label to give a direction to the edge connecting them

(call these *E-edges*). We have the following constraints on vertices with s-label E: a defect can be labeled E; a corner can be labeled E iff it has exactly one E-edge directed in; any other vertex can be labeled E iff it has exactly one E-edge directed in and exactly one E-edge directed out. If a corner has s-label E, then its neighborhood does not have to have senses of direction.

Suppose that M does not halt and that the graph is a sufficiently large 4-regular grid. It is clear that no corner can be labeled E since the graph has no defects. The labeling could contain cycles of vertices labeled E, but this will violate other constraints if it occurs in the active region. It then follows as above that a legal labeling exists, but cannot be found in constant time.

If the graph is not a 4-regular grid then it must have a defect. In this case, it can be seen that the entire graph can be s-labeled with E's and I's. Say that the defect u *kills* the corner w if there is a directed path of E-labeled vertices from u to w. By choosing the constant c above large enough, it can be seen that there is an E-labeling such that each defect kills at most one corner and E-labeled vertices in different paths are not adjacent.

Suppose that M halts in t steps. Assume for the moment that no two corners of the graph are within distance $4t + 4$ of each other. Consider the following labeling on one level. Fix some corner w. Say that w is a *good corner* if $B(w, 2(t + 1))$ contains no defects (i.e., looks like part of a 4-regular grid). If w is good, then the s-labeling of the appropriate $(t+1)\times(t+1)$ subgrid describes a computation of M as in part (1). If $B(w, 2(t + 1))$ contains a defect u, then we choose in some systematic way a path labeled E from one such defect u to w. The rest of $B(w, 2(t + 1))$ is labeled I. Any vertex not within distance $2(t + 1)$ from some corner is labeled I. Such a labeling can be found in time $O(t)$. If corners can be close together, it must be checked that four levels are enough to do the labeling. Define a graph where there is a vertex for every good corner and an edge connecting two vertices if the computations started at the corresponding good corners overlap. It can be checked that no component of this graph has more than four vertices, so the labeling can be done on four levels and a local algorithm can determine an assignment of good corners to levels.

For the case $d = 3$ we use, instead of two-dimensional grids, degree-3 "honeycomb" graphs; these look like a tiling of the plane with hexagons. For $d \geq 5$, we can use $d - 3$ copies of the same grid graph, where corresponding vertices in different copies are connected as a clique. □

Remark. The LCLs constructed in the proof above have an upper bound r_0 on their radius, where r_0 is a constant; i.e., it does not depend on the machine M. It follows that there is an infinite time hierarchy of LCLs with some fixed radius r_0. That is, for every time t, there is an LCL of radius r_0 that cannot be solved by any local algorithm with time bound t, but can be solved by some local algorithm with some time bound $t' > t$.

In contrast, the following holds for degree 2.

THEOREM 4.2. *Let $\mathcal{G}$ be the class of 2-regular graphs or the class of graphs of maximum degree 2. $Y(\mathcal{G})$ and $N(\mathcal{G})$ are recursively separable. Moreover, this can be done in time polynomial in the size of the input $\mathcal{L}$.*

Proof. Consider first the case of 2-regular graphs. Let $\mathcal{L}$ be a given LCL and r be its radius. If (H, s) belongs to the set $\mathcal{C}$ of locally consistent labelings, then either H is a simple path of $2r + 1$ vertices with s at the center or H is a ring having at most $2r$ vertices. Assuming that every graph in $\mathcal{G}$ has an $\mathcal{L}$-legal labeling, we claim that there is a local algorithm with some constant time bound t that solves $\mathcal{L}$ for $\mathcal{G}$ iff $\mathcal{C}$ contains a line segment in which all vertices are labeled the same, say α. The "if" direction is obvious, since any ring having at least $2r + 1$ vertices has an $\mathcal{L}$-legal labeling where all vertices are labeled α. (In this case, a local algorithm with time bound r checks whether it is working on a cycle of size at least $2r + 1$. If so, it produces the label α. If not, it knows the entire graph so it can use the rank of

the ID of the vertex to find a label for the vertex by looking in a table, where the table contains an $\mathcal{L}$-legal labeling for each cycle of size less than $2r + 1$.) For the "only if" direction, by Theorem 3.3 there is an order-invariant A' with time bound t that solves $\mathcal{L}$ for $\mathcal{G}$. Consider the ID numbering of a ring where the order of the ID's increase around the ring, except at one point where the ordering wraps around. If the ring is sufficiently large, there will be a segment of length $2r + 1$ such that A' gives the same label to every vertex of the segment.

The case of maximum degree 2 is a little more complicated. It is still necessary that $\mathcal{C}$ contain a line segment labeled the same, but this is no longer sufficient. Since the graph could be a line, we must also check that there is an $\mathcal{L}$-legal labeling in which all vertices, except possibly vertices within some bounded distance from the endpoints of the line, are labeled the same. This is easily reduced to a reachability problem on a certain graph K. Fix a left-to-right orientation of a line. For every member of $\mathcal{C}$ that is a line segment, there are two vertices in K, one for each left-to-right orientation of the segment. There is an edge directed from v to w iff there is a labeling of the oriented line in which the center of v is just to the left of the center of w. For example, if $r = 2$, if v is the locally consistent labeling B-C-D, where the center is the vertex labeled B, and if w is B-C-D-X, where the center is labeled C, then there is an edge from v to w. Any vertex such as v for which the center is the leftmost endpoint of the segment is called a *source*. A goal vertex is any vertex corresponding to a line segment containing $2r + 1$ vertices all labeled the same. Then there is a local algorithm with some constant time bound t iff there is a directed path from some source to some goal. Note that if there is such a path then there is one of length $O(|\mathcal{C}|)$, so $t = O(\max\{|\mathcal{C}|, r\})$ suffices. □

Remark. It can also be shown, for the graph classes $\mathcal{G}$ in Theorem 4.2, that it is decidable for a given LCL $\mathcal{L}$ whether every graph in $\mathcal{G}$ has an $\mathcal{L}$-legal labeling.

The final result of this section is an easy consequence of Theorem 3.3.

THEOREM 4.3. *Fix any $d \geq 2$ and let $\mathcal{G}$ be the class of d-regular graphs or graphs of maximum degree d. It is decidable, given $\mathcal{L}$ and t, whether there is a local algorithm with time bound t that solves $\mathcal{L}$ for $\mathcal{G}$.*

Proof. By Theorem 3.3 we can restrict attention to order-invariant algorithms with time bound t. There is only a finite number of such algorithms, and for each algorithm A we can test whether it solves $\mathcal{L}$ for $\mathcal{G}$ as follows. Let r be the radius of $\mathcal{L}$. Let $\mathcal{H}$ be the set of centered graphs (H, s) such that, for some $G \in \mathcal{G}$ and some vertex v of G, H is isomorphic to $B_G(v, r + t)$ under an isomorphism mapping s to v. For the classes $\mathcal{G}$ under consideration, there are a finite number of such graphs for each r and t, and there is an algorithm that lists them given r, t. For each $(H, s) \in \mathcal{H}$ and each order-inequivalent ID numbering of the vertices of H, we apply A to each vertex in $B_H(s, r)$ to obtain a labeling of $B_H(s, r)$. If this labeling is not consistent according to $\mathcal{L}$, then A is not correct, since A will fail at some vertex v of some $G \in \mathcal{G}$, where $B_G(v, r + t)$ is isomorphic to H; by "fail" we mean that the labeling of $B_G(v, r)$ is not consistent according to $\mathcal{L}$. On the other hand, if A always labels $B_H(s, r)$ correctly for every H and every numbering, then A is correct. For if A fails at some vertex v of some $G \in \mathcal{G}$, then A fails at v in $B_G(v, r + t)$, which is isomorphic to some H. □

5. Randomized algorithms. We now turn to randomized algorithms and show, for certain classes of graphs, that randomization does not help in solving LCLs in constant time. A *randomized local algorithm* P with time bound t is specified by a deterministic local algorithm A with time bound t and a function $b(n)$ to positive integers called the *randomization bound.* In this case, A expects each vertex to be labeled with an input label (if the LCL has input labels), an ID number, and a random number. To run P on a graph G that is ID-numbered and input-labeled, first randomly and independently choose for each vertex a random number in the range $[1, b(l)]$, where l is the largest ID in G; then run A on the resulting graph. We assume no upper bound on the growth rate of $b(n)$. We say that *P solves $\mathcal{L}$ for $\mathcal{G}$ with error*

probability ε if, for every input-labeled and ID-numbered $G \in \mathcal{G}$, P produces an $\mathcal{L}$-legal labeling with probability at least $1 - \varepsilon$.

Remark. The above definition of a randomized local algorithm might seem too liberal, since it allows the range of randomization at a particular vertex v to depend on the largest ID in the entire graph. It would be more reasonable to have the range of randomization at v depend only on v's ID. But since our result is that randomization does not help in solving labeling problems locally, there is no harm in using the more liberal definition. On the other hand, this definition may seem too restrictive, since the definition of success is global. However, in the deterministic case we wanted the labeling to be legal everywhere, not just in most vertices. Indeed, if all we require is that, for each vertex v, the probability that v is legally labeled be at least $1 - \varepsilon$, then randomization does help; consider the problem of 3-coloring in a ring. This problem has no deterministic local algorithm [10] nor a probabilistic one [14] (with the global correctness requirement). Suppose that we start with all vertices uncolored and at every step each vertex that is not permanently colored chooses a random color. If the vertex chose a color different from the colors of its two neighbors, then this color is considered permanent. If this algorithm is executed for t steps, then we can say that, for each vertex v, the probability that v is legally colored is at least $1 - \varepsilon$, where ε decreases exponentially in t.

THEOREM 5.1. *Fix an LCL $\mathcal{L}$ and a class $\mathcal{G}$ of graphs closed under disjoint union. If there is a randomized local algorithm P with time bound t that solves $\mathcal{L}$ for $\mathcal{G}$ with error probability ε for some $\varepsilon < 1$, then there is a deterministic local algorithm A with time bound t that solves $\mathcal{L}$ for $\mathcal{G}$.*

Proof. Suppose for contradiction that there is no deterministic local algorithm with time bound t that solves $\mathcal{L}$ for $\mathcal{G}$. In particular, there is no order-invariant algorithm with this property. There is an upper bound on the number of order-invariant local algorithms with time bound t (where the upper bound depends only on $\mathcal{L}$, t, and the degree bound d). This immediately proves the following claim.

CLAIM 5.1. *There is a number N such that every order-invariant local algorithm A' with time bound t fails on some particular input-labeled and ID-numbered graph G having at most N vertices, where by "fail" we mean that A does not produce an $\mathcal{L}$-legal output labeling.*

Let R be the number given by Lemma 3.2. Let m be the minimum number of vertices in a graph in $\mathcal{G}$. If $R < N + m$, then take $R = N + m$ to ensure that $R \geq N + m$. For $j \geq 1$, let $S_j = \{(j-1)R + 1, \ldots, jR\}$. Let $\mathcal{I}_j$ be the set of graphs $G \in \mathcal{G}$ having at most N vertices that are input-labeled and have ID numbering drawn from S_j. If Σ is the input alphabet, an upper bound on the cardinality of $\mathcal{I}_j$ is

$$k = 2^{\binom{N}{2}} R^N |\Sigma|^N.$$

Choose q large enough that $(1 - \frac{1}{k})^q < 1 - \varepsilon$.

The key to the proof is the following claim.

CLAIM 5.2. *For every j with $1 \leq j \leq q$, there is a graph $G_j \in \mathcal{I}_j$ such that, if P is run on G_j with randomization bound $b(qR)$, then the probability that P fails on G_j is at least $1/k$.*

To prove the claim, suppose it is false, i.e., that for every $G \in \mathcal{I}_j$, P fails with probability strictly less than $1/k$. We can view a random choice of P as a sequence of random numbers $\rho_1, \ldots, \rho_R$ in the interval $[1, qR]$, where ρ_i is the random number chosen for the vertex with ID $(j-1)R + i$. Since the error probability is less than the reciprocal of the number of graphs in $\mathcal{I}_j$, it follows by a standard argument (e.g., [1]) that there must be a particular choice $\hat{\rho}_1, \ldots, \hat{\rho}_R$ such that P is correct on all graphs in $\mathcal{I}_j$ when this particular random choice is made. We can then obtain a deterministic algorithm A that works on $\mathcal{I}_j$. This algorithm first chooses the random number $\hat{\rho}_{i-(j-1)R}$ at the vertex with ID i for every i and then simulates P.

By Lemma 3.2, there is an order-invariant A' that is correct on all of $\mathcal{I}_j$, but this contradicts Claim 5.1, and so proves Claim 5.2.

It is now easy to complete the proof of Theorem 5.1. Run P on the graph G consisting of the disjoint union of the G_j for $1 \leq j \leq q$. (If no vertex of this graph has label qR, then add another component with m vertices and maximum ID label qR. This is possible since $qN + m \leq qR$.) Since P fails independently with probability at least $1/k$ on each G_j, it follows from the choice of q that P fails on G with probability strictly greater than ε. This contradiction proves the theorem. □

A version of Theorem 5.1 holds also for certain classes of connected graphs, for example, connected d-regular graphs and connected graphs of maximum degree d for any fixed $d \geq 2$. All we need is the ability to connect the graphs $G_1, \ldots, G_q$ into a single graph in the class in such a way that P's error probability on each piece does not decrease when the pieces are connected. For example, the following theorem holds.

THEOREM 5.2. *Fix an LCL $\mathcal{L}$ and a $d \geq 2$ and let $\mathcal{G}$ be the class of connected d-regular graphs or the class of connected graphs of maximum degree d. If there is a randomized local algorithm P with time bound t that solves $\mathcal{L}$ for $\mathcal{G}$ with error probability ε for some $\varepsilon < 1$, then there is a deterministic local algorithm A with time bound $2(t + r) + 1$ that solves $\mathcal{L}$ for $\mathcal{G}$, where r is the radius of $\mathcal{L}$.*

Proof. The proof is very similar to the previous one, and we only sketch the differences. Claim 5.1 is modified to state that there is an N such that every order-invariant local algorithm with time bound t fails on some connected graph with radius at least $t + r + 1$ and at most N vertices. If the modified claim were not true, there would be a local algorithm with time bound $2(t + r) + 1$ that solves $\mathcal{L}$ for $\mathcal{G}$. This algorithm first checks whether it is working on a graph of radius at most $t + r$ by trying to inspect the entire graph. If so, it can produce a labeling because it knows the entire graph. If not, it recourses to an algorithm with time bound t that works on every connected graph of radius at least $t + r + 1$. The set $\mathcal{I}_j$ now contains pairs (G, v), where G has at most N vertices and radius at least $t + r + 1$ and v is a vertex of G, so the bound k increases by a factor of N. Now the conclusion of Claim 5.2 is that for every j there is a $(G_j, v_j) \in \mathcal{I}_j$ such that, with probability at least $1/k$, P fails on G_j at the particular vertex v_j, meaning that $B(v_j, r)$ is not labeled correctly. Since the radius of G_j is at least $t + r + 1$, there is some edge e such that removing e does not affect the behavior of P when labeling $B(v_j, r)$. Let G'_j be the graph with this e removed. We can now connect $G'_1, \ldots, G'_q$ to a graph in the class $\mathcal{G}$, using the endpoints of the removed edges as connection points. □

Remark. An alternate conclusion in Theorem 5.2 is that there is a deterministic local algorithm A with time bound t that solves $\mathcal{L}$ for all graphs in $\mathcal{G}$ having radius at least $t + r + 1$.

6. Weak coloring. We now describe a locally checkable labeling problem that can be solved locally in graphs containing only vertices of odd degree. A weak c-coloring of a graph is an assignment of numbers from $\{1, \ldots, c\}$ to the vertices of the graph such that for every nonisolated vertex v there is at least one neighbor w such that v and w receive different colors. Clearly weak c-coloring of a graph of degree at most d is an LCL problem of radius 1.

It is not hard to see that every graph has a weak 2-coloring: consider a breadth-first spanning tree of the graph. Assign one color to the even levels and a different color to the odd levels. However, this particular coloring cannot be computed locally. As we shall see, if all the vertices of the graph have odd degree, then it is possible to find a weak 2-coloring. As far as we know this is the first nontrivial LCL problem that has been shown to have a local algorithm. However, if the degree is even then it is impossible to compute such a coloring or any weak c-coloring for a fixed c locally.

6.1. Weakly coloring graphs of odd degree. We describe a way of finding a weak coloring in odd-degree graphs. We first show a two-step method for weak $d(d+1)^{d+2}$-coloring and then show how to reduce the number of colors to two using additional steps.

Consider first the case of a d-regular graph, where d is odd and $d \geq 3$. For a vertex v let $\mathrm{ID}(v)$ be the ID number assigned to v. We denote the color of a vertex v by a vector $C_v = \langle C_v[0], C_v[1], \ldots, C_v[d+1] \rangle$, where each component is in $\{1, \ldots, d+1\}$. The following procedure is used at vertex v:

1. Get $\mathrm{ID}(w)$ for all neighbors w of v. Sort the set of IDs of neighbors including $\mathrm{ID}(v)$. Let $r_v(w)$ denote the rank of $\mathrm{ID}(w)$ among the neighborhood of v (where the neighborhood of v includes v itself). For definiteness, say that the smallest ID has rank 1, the second smallest has rank 2, etc. Let $C_v[0]$ be $r_v(v)$.
2. Get $r_w(v)$ from each neighbor w, i.e., the rank of $\mathrm{ID}(v)$ among the neighborhood of w. Set $C_v[r_v(w)] = r_w(v)$.

CLAIM 6.1. *The coloring achieved by this algorithm is a legal weak coloring if d is odd.*

Proof. Consider a vertex v. If not for all neighbors w of v we have $r_w(w) = r_v(v)$, then we are done, since there will be a neighbor w of v such that the color of w differs from the color of v in the first component. Otherwise, there are two cases. In the first case, assume that $1 \leq r_v(v) \leq \frac{d+1}{2}$. This means that there are $d + 1 - r_v(v) \geq \frac{d+1}{2}$ neighbors w such that $\mathrm{ID}(w) > \mathrm{ID}(v)$. For each of them $r_w(v) < r_v(v)$, since $r_w(w) = r_v(v)$. Therefore, by the pigeonhole principle there are two neighbors w and x such that $r_w(v) = r_x(v) = j$. Hence

$$C_w[j] = C_w[r_w(v)] = r_v(w) \neq r_v(x) = C_x[r_x(v)] = C_x[j].$$

$C_w[j] \neq C_x[j]$ means that v has two neighbors with two different colors, one of which must be different than C_v. Similarly, in the other case, if $\frac{d+1}{2} + 1 \leq r_v(v) \leq d + 1$, then there are $r_v(v) - 1 \geq \frac{d+1}{2}$ neighbors w such that $\mathrm{ID}(w) < \mathrm{ID}(v)$. For each of them, $r_w(v) > r_v(v) \geq \frac{d+1}{2} + 1$, and a pigeonhole argument again shows that there must be two neighbors that are colored with two different colors. □

If vertices can have different (odd) degrees, we can simply add another component to C_v which contains the degree of v. If v has a neighbor with a different degree, then it has a neighbor with a different color; otherwise, Claim 6.1 applies.

To go from $d(d+1)^{d+2}$ colors to two colors we employ two kinds of color reductions: one is a Cole–Vishkin [6] style that allows us to cut the number of colors logarithmically in every round, but seems to have its limit at four. The other method allows us to reduce the number of colors one at a time.

The Cole–Vishkin style method is as follows: Suppose that we have a legal weak coloring with c colors and let c' be the smallest integer such that $\binom{c'}{\lfloor c'/2 \rfloor} \geq c$. Associate with every $i \in \{1, \ldots, c\}$ a different subset $S_i \subset \{1, \ldots, c'\}$ of size $\lfloor c'/2 \rfloor$. (Such an assignment is a Sperner system, i.e., no subset is contained in another.) We can reduce the number of colors from c to c' in one round. Every vertex v colored i finds a neighbor colored j such that $j \neq i$. There must be an element $x \in S_i$ such that $x \notin S_j$. x is v's next color. It is easy to see that this method preserves weak coloring and reduces the number of colors by almost a logarithmic factor per round. More precisely, the number c' of colors after a step of the reduction is related to the number c before the reduction by $c' = \log c + O(\log \log c)$, where logarithms are to the base 2. (This is (almost) a bit more efficient than the original Cole–Vishkin reduction, where the relation in our case is $c' = 2\lceil \log c \rceil$.) The method is applicable as long as $c > 4$. A simple calculation (cf. [8, p. 437]) shows that a weak 4-coloring is found after $\log^* d + a$ rounds for some constant a. (Another way to see this is to note that the base of the logarithm affects the expression $\log^* d + a$ only in the additive term a.)

When we are stuck (i.e., $c = 4$) we can recourse to the following reduction from a weak coloring with c colors $\{1, 2, \ldots, c\}$ (called the original coloring) to one with 2 colors $\{0, 1\}$ (called the recoloring). The recoloring is done in c rounds. At the ith round, every vertex with original color i recolors itself according to the following rules:

1. If v has original color i and all neighbors of v have original color $\geq i$, then v recolors itself 0.
2. Otherwise, v must have at least one neighbor with original color smaller than i, so it has at least one neighbor that has recolored itself at an earlier round. If all the recolored neighbors of v have color 1, then v recolors itself 0. Otherwise (v has at least one neighbor recolored 0), v recolors itself 1.

It is easy to verify that this yields a weak 2-coloring. Every v that recolors itself using the second rule clearly has a neighbor recolored differently. Suppose that v recolors itself 0 using the first rule. Then it must have a neighbor w with original color $j > i$. Then w will recolor itself using the second rule during round j, and it will recolor itself 1 since it has a neighbor (namely, v) recolored 0 at an earlier round (namely, i). We therefore get the following theorem.

THEOREM 6.1. *Let $\mathcal{O}_d$ be the class of graphs of maximum degree d where the degree of every vertex is odd. There is a constant b such that, for every d, there is a local algorithm with time bound $\log^* d + b$ that solves the weak 2-coloring problem for $\mathcal{O}_d$.*

Remark. In §7 we will want to apply the weak coloring algorithm to graphs that may have vertices of even degree, and we will use the following additional property of the algorithm. Say that v is properly colored if it has at least one neighbor colored differently. Suppose that the weak 2-coloring algorithm is applied to an arbitrary (bounded degree) graph G. If v is not properly colored then (1) the degree d of v is even, (2) its rank $r_v(v)$ in its neighborhood is $d/2 + 1$, and (3) every neighbor w of v has degree d and rank $r_w(w) = d/2 + 1$ as well. To see that these properties hold, consider first the coloring produced by the initial two-step algorithm. Properties (1) and (2) follow since our proof of Claim 6.1 shows that v is properly colored if either $r_v(v) \leq \frac{d+1}{2}$ or $r_v(v) \geq \frac{d+1}{2} + 1$. The only other possibility is that d is even and $r_v(v) = d/2 + 1$. Since the color of a vertex u contains its degree and its rank $r_u(u)$, (3) is obvious. It is also easy to check that both of the color reduction methods preserve proper coloring, i.e., if v is properly colored before a reduction, then it is properly colored after the reduction.

To close this section we note that there is no one-step method for finding a weak c-coloring.

THEOREM 6.2. *For any constants c and $d \geq 2$, there is no local algorithm with time bound 1 that solves weak c-coloring for the class of d-regular graphs.*

Proof. By Theorem 3.3, if there were such a local algorithm A, there would be an order-invariant one A', also with time bound 1. Consider any d-regular graph that contains a vertex v such that $B(v, 2)$ (the neighborhood of radius 2 around v) is a tree of height 2 rooted at v. Number the vertices of $B(v, 2)$ with IDs so that $r_v(v) = 2$ and $r_w(w) = 2$ for every neighbor w of v (it is easy to see that this can be done). Then A' assigns the same color to v and all its neighbors. □

6.2. Impossibility of weak coloring graphs of even degree. In this section we note that it is impossible in general to weakly color all graphs with even degree. In particular we show that for any c and k it is impossible to weakly c-color any class of graphs that contains the k-dimensional meshes. The vertex set of a k-dimensional mesh is $\{0, 1, \ldots, m\}^k$ for some m, and two vertices are connected by an edge if the L_1-distance between them is 1. A k-dimensional mesh has (some) vertices of even degree $d = 2k$.

THEOREM 6.3. *For any c, k, and t, there is no local algorithm with time bound t that solves the weak c-coloring problem for the class of k-dimensional meshes.*

Proof. Theorem 3.3 says that if there exists a local algorithm for an LCL problem then there is one that uses only the relative order of the IDs. For a vertex v of a mesh M, let $R_M(v, t)$ be the graph $B_M(v, t)$ (the neighborhood of radius t around v) where each vertex u is labeled with the rank of its ID among the IDs in $B_M(v, t)$. By Theorem 3.3 it is sufficient to come up with a way to assign IDs to vertices such that for any t there will be a k-dimensional mesh M and a vertex v such that, for all neighbors u of v, $R_M(v, t)$ and $R_M(u, t)$ are the same (i.e., v and its neighbors see the same relatively ordered t-neighborhood). However, if M has diameter at least $2(t + 1)$ then it possible to achieve such an ID assignment: consider the coordinates of a vertex and say that vertex u is larger than v if the lexicographical order of the coordinates of u is larger than that of v. It is clearly possible to assign IDs such that $\mathrm{ID}(u) > \mathrm{ID}(v)$ iff u is larger than v. Hence any vertex that is of distance at least $t + 1$ from every boundary of the mesh has the property we are after. □

The same result holds for a class of $(2k)$-regular graphs, the k-dimensional analogue of torus graphs.

A consequence of this result is that if we extend the definition of weak coloring so that each vertex v must have at least two neighbors colored differently than v (call this *2-weak c-coloring*), then for every fixed d and c a coloring cannot be found in constant time for d-regular graphs even if d is odd. The reasoning is as follows: Given any $(2k)$-regular graph G, form a $(2k + 1)$-regular graph G' by taking two copies of G with each pair of corresponding vertices connected by an edge. The IDs in one copy are all chosen to be larger than the IDs in the other copy, but so the two copies appear identical with respect to the relative order within a copy. A 2-weak c-coloring of G' immediately gives a weak c-coloring in each copy of G. Given a local algorithm that finds a 2-weak c-coloring in graphs of odd degree $2k + 1$, we therefore obtain a local algorithm that finds a weak c-coloring in graphs of even degree $2k$.

7. A locally solvable resource allocation problem. We show how to solve the formal-dining philosophers problem mentioned in the introduction. What we assume about the underlying graph is that the minimum degree is three. (If the minimum degree is two, then we cannot hope to solve it locally, as we argue below.)

We first start with a coloring with three colors $\{0, 1, *\}$ with the following property: all vertices colored $c \in \{0, 1\}$ have at least one neighbor colored $1 - c$. If v is colored with a $*$ or if any of the neighbors of v is colored with a $*$, then the degree of v is even and half the neighbors of v have an ID smaller than $\mathrm{ID}(v)$. This coloring is a product of the method described in §6.1. Suppose that we run the algorithm described there. Since we do not assume here that every vertex has odd degree, the algorithm could *fail* at some vertices v, meaning that all the neighbors of v are colored the same as v. Suppose that if the algorithm fails at v, then v recolors itself with a $*$. By the remark following Theorem 6.1, the coloring fails at v only when the degree d of v is even, its rank $r_v(v)$ among its neighbors is $d/2 + 1$, every neighbor w of v has degree d, and $r_w(w) = d/2 + 1$ as well.

The algorithm for the formal-dining philosophers problem is a combination of two algorithms: one for the problem on graphs that are weakly 2-colored and the other for the case where half the neighbors of a vertex have a smaller ID. The vertices colored $*$ essentially grab two of the adjacent cuff links permanently. More precisely, a vertex u colored $*$ picks two neighbors v and w such that $\mathrm{ID}(u) > \mathrm{ID}(v)$ and $\mathrm{ID}(u) > \mathrm{ID}(w)$, and assigns the cuff links on (u, v) and (u, w) to u permanently. After this we have that every vertex with color $c \in \{0, 1\}$ still has at least two nonassigned edges adjacent to it and at least one of its neighbors has color $1 - c$. This is true since if it is a neighbor of a $*$, then its degree is even (at least 4) and its rank among its neighbors is half the degree plus one. Hence at most half of its adjacent edges are grabbed permanently. Unlike the $*$ colored vertices, the $\{0, 1\}$ colored vertices must run a dynamic algorithm in order to get cuff links.

As for the $\{0, 1\}$ colored vertices, it is convenient for the exposition to assume first that we have a coloring of the graph with the property that every vertex has at least one neighbor colored 0 and at least one neighbor colored 1.[2] We will later remove this assumption. As a preliminary step, every vertex colored $c \in \{0, 1\}$ selects a particular neighbor colored c as its "first neighbor" and a particular neighbor colored $1 - c$ as its "second neighbor." When we say that a vertex p "requests" a cuff link, we mean that it tries to grab the cuff link; if the other vertex q sharing this cuff link currently has it, then p waits for q to release it. Now the protocol for a vertex colored $c \in \{0, 1\}$ is as follows:

1. Request cuff link from the first neighbor (colored c).
2. Request cuff link from the second neighbor (colored $1 - c$).
3. Eat.
4. Release cuff links.

CLAIM 7.1. *The maximum length of a waiting chain in the above protocol is* 2.

Proof. If a vertex is waiting at Step 1, then the vertex it is waiting for must be at least at Step 2. If a vertex v is waiting for its second neighbor w at Step 2, then v and w are colored differently, which means that v is the second neighbor of w. So w must be at Step 3 or 4. □

Suppose now that all we can say is that a vertex colored $c \in \{0, 1\}$ has at least one neighbor colored $1 - c$, i.e., all its neighbors might be colored $1 - c$. If at Steps 1 and 2 arbitrary neighbors colored $1 - c$ are approached, then we are not guaranteed to be deadlock free anymore. The selection of second neighbors should be done in a way that does not induce long "neighborly" chains. To this end, we differentiate between the vertices colored 0 and 1. Each vertex colored 1 chooses a particular neighbor colored 0 as its second neighbor. These choices are announced to their neighbors. A vertex u colored 0 waits to hear whether it has been chosen as the second neighbor by any of its neighbors. If it has, then it tries to match their choices; i.e., if any of u's neighbors has designated it as a second neighbor, u picks it (or one of them in case there are several) as u's second neighbor. Otherwise, u chooses an arbitrary neighbor colored 1 as its second neighbor. Each vertex colored 0 or 1 then chooses an arbitrary neighbor, other than its second neighbor, to be its first neighbor. (Of course, u should not choose a neighbor w colored $*$ if w has permanently grabbed the cuff link on the edge (u, w). On the other hand, if w is colored $*$ and has not grabbed the cuff link then it never will, so u can choose this w, and u will never have to wait for w.)

CLAIM 7.2. *Given any assignment of first and second neighbors consistent with the above description, the maximum length of a waiting chain is at most* 4.

Proof. A configuration that the preliminary step as described above assures won't occur is as follows: three vertices w_1, w_2, and w_3 colored 1, 0, and 1, respectively, such that w_2 is the first neighbor of w_3, w_2 is the second neighbor of w_1, and w_3 is the second neighbor of w_2. This cannot occur since w_2 was chosen to be a second neighbor of at least one vertex (namely, w_1), but w_2 is not the second neighbor of w_3. Hence w_2 would not choose w_3 as its second neighbor. Now consider a contradiction to the claim, i.e., six vertices $u_0, u_1, u_2, u_3, u_4, u_5$ such that each u_i is waiting for u_{i+1} for $0 \leq i \leq 4$. Let c be the color of u_1. Since u_1, u_2, u_3, and u_4 are waiting at Step 2, it must be the case that u_2 is colored $1 - c$, u_3 is colored c, u_4 is colored $1 - c$, and u_5 is colored c. Also for $1 \leq i \leq 3$ we have that u_i is the first neighbor of u_{i+1}, and for $1 \leq i \leq 4$ we have that u_{i+1} is the second neighbor of u_i. Therefore, if $c = 0$ then the trio $\{u_2, u_3, u_4\}$ constitutes a forbidden configuration, and if $c = 1$ then $\{u_1, u_2, u_3\}$ constitutes a forbidden configuration.

(We remark that a waiting chain of length 4 can occur.) □

[2] Although by the Lovász Local Lemma such a coloring exists in regular graphs with sufficiently large degrees, it is impossible to find such a coloring locally even in odd-degree graphs.

Since this argument remains valid if some of the u_i are the same, the argument shows that a deadlock (a waiting cycle) cannot occur, since a waiting cycle would produce, in effect, a waiting chain of arbitrary length.

Therefore the combined protocol is:

- Run the coloring algorithm of §6.1 resulting in a $\{0, 1, *\}$ coloring.
- All the vertices colored $*$ permanently grab two cuff links as described above.
- All $\{0, 1\}$ colored vertices find first and second neighbors as described above.
- When a vertex becomes hungry, then if it is colored $*$ it simply uses its permanently assigned cuff links. Otherwise it runs the protocol above.

THEOREM 7.1. *The above protocol solves the formal-dining philosophers problem locally.*

A consequence of bounded-length waiting chains is that the failure locality of the protocol is constant. As defined by Choy and Singh [5], a protocol has *failure locality* l if every vertex v remains starvation free even if processors at distance larger than l from v fail. Another consequence is that the response time of the protocol is constant. This means that, for every upper bound ν on the message delivery time between adjacent vertices and every upper bound τ on the time that a vertex can remain in the eating state before entering the resting state, there is a $\rho = \rho(\nu, \tau)$ such that ρ is an upper bound on the time that a vertex can remain in the hungry state before entering the eating state. We now justify our requirement that the conflict graph have minimum degree 3, by arguing that if the conflict graph is a ring then the formal-dining philosophers problem (which is the same as the usual-dining philosophers problem in this case) cannot be solved locally, meaning in particular that the response time is constant. This follows from a more general result about the local unsolvability of certain dining philosophers problems where the condition under which a philosopher can eat is a threshold condition on the number of resources owned. For constants d and m with $d \geq 2$ and $m \leq d$, define the (d, m)*-dining philosophers problem* as follows: the conflict graph has minimum degree d, and in order to eat, a philosopher must own the resources on any m incident edges.

THEOREM 7.2. *If* $m \geq \lceil d/2 \rceil + 1$, *there is no algorithm with constant response time for the* (d, m)*-dining philosophers problem.*

Proof. It suffices to prove this for every even d. Fix some $d = 2k$. We assume that the interconnection graph is the same as the conflict graph, a k-dimensional torus (i.e., a k-dimensional mesh with additional wrap-around edges to produce a d-regular graph). We first show that a solution to the (d, m)-dining philosophers problem with constant response time would give a local solution to the following LCL problem, for some constant p depending only on d:

1. Each vertex is labeled either 1 or 2.
2. Each k-dimensional $p \times p \times \cdots \times p$ submesh M contains some vertex labeled 1 and some vertex labeled 2.

Given an arbitrary ID-numbered torus, run the assumed (d, m)-dining philosophers algorithm starting in the configuration where all vertices are initially hungry. Let the vertices operate in lock-step synchrony, i.e., each message delay is one time unit. When a vertex enters the eating state, let it remain in the eating state for one time unit, and remain in the resting state thereafter. So $\nu = \tau = 1$. The following rules are used to determine the label of vertex v. Let s be the step at which v enters the eating state. If v has not received the message "eating" from one of its neighbors at step s or earlier, then v chooses the label 1 and sends the message "eating" to all of its neighbors at step s (in addition to any messages that the (d, m)-dining philosophers algorithm sends at this step); the "eating" message is received by v's neighbors at the next step $s + 1$. Otherwise (v received an "eating" message at step s or earlier), v chooses the label 2 and does not send an "eating" message. The labeling algorithm runs in constant time $\rho(1, 1)$. It remains to show that condition 2 above holds for a large enough p. Assume

that $p \geq 3$. If there is a $p \times \cdots \times p$ submesh M with all vertices labeled 2, then there would be some v such that v and all of its neighbors are labeled 2. But this is impossible since, if all the neighbors of v are labeled 2, they do not send "eating" to v, so v will be labeled 1 at the step when it eats. If there is a $p \times \cdots \times p$ submesh M with all vertices labeled 1, then all vertices of M enter the eating state at the same step s. This gives a contradiction as follows: The number of edges incident on vertices of M is at most $(d/2)p^k + O(p^{k-1})$, but in order for each vertex of M to own at least m of these edges, there must be at least mp^k of them. Since $m \geq d/2 + 1$ by assumption, we obtain a contradiction by choosing p large enough.

It is now easy to prove that this LCL cannot be solved locally for k-dimensional torus graphs. The proof is similar to that of Theorem 6.3. As before, it suffices to consider order-invariant algorithms. For the ID-numbering described in the proof of Theorem 6.3, every sufficiently large torus will have a $p \times \cdots \times p$ submesh M such that every vertex of M receives the same label. But this contradicts the definition of the LCL. □

In particular, the (d, d)-dining philosophers problem is not solvable with constant response time for any $d \geq 2$.

Mayer, Naor, and Stockmeyer [12] have proved the converse of Theorem 7.2: if $m \leq \lceil d/2 \rceil$, then there is a protocol with constant response time for the (d, m)-dining philosophers problem.

8. Open questions. This is an early attempt to study what can and cannot be computed locally, and many questions remain open. A general direction for future work is to obtain more information about what sorts of labeling problems and resource allocation problems can be solved locally. In particular, the following specific questions are suggested by our work.

1. Consider the problem of assigning an orientation to some edges of the graph so that every vertex has either no edge directed in or two edges directed in, and the assignment is maximal with respect to the number of vertices that have two edges directed in. This problem (the *maximal in-degree 2 problem*) was suggested by the formal-dining philosophers problem. In the case in which all philosophers are initially hungry, such an orientation corresponds to an assignment of cuff links that is maximal with respect to the number of philosophers who are eating. We can show that this problem cannot be solved in constant time for d-regular graphs where $d \leq 4$. Can it be solved in constant time for some $d \geq 5$?

2. We have shown that a weak 2-coloring can be found in time $O(\log^* d)$ in odd-degree graphs of maximum degree d. Is this the best possible time as a function of d, or is there some fixed time t that is sufficient for all d?

3. We have shown that a weak c-coloring cannot be found in constant time for certain graphs having vertices of even degree (meshes). Does the same hold for trees where every nonleaf has even degree? We conjecture that the result holds for any class of even-degree edge-transitive graphs. Is this conjecture true?

Some other questions are addressed in [12].

1. The locality framework is extended to dynamic graphs, where edges can fail and later recover and new nodes and edges can be added to the graph.

2. The amount of initial symmetry-breaking needed to solve certain problems locally is investigated.

Acknowledgments. We thank Ron Fagin for discussions regarding the relevance of locality in first-order logic, Shay Kutten for pointing out the relevance of locality in work on self-stabilization, and Seffi Naor for suggesting the formal-dining metaphor.

REFERENCES

[1] L. ADLEMAN, *Two theorems on random polynomial time*, in Proc. 19th IEEE Symposium on Foundations of Computer Science, Ann Arbor, MI, 1978, pp. 75–83.

[2] Y. AFEK, S. KUTTEN, AND M. YUNG, *Local detection for global self stabilization*, manuscript; a preliminary version appeared in Proc. 4th Workshop on Distributed Algorithms, Lecture Notes in Computer Science, 486, Springer-Verlag, New York, 1991, pp. 15-28.

[3] B. AWERBUCH, A. V. GOLDBERG, M. LUBY, AND S. A. PLOTKIN, *Network decomposition and locality in distributed computation*, in Proc. 30th IEEE Symposium on Foundations of Computer Science, Research Triangle Park, NC, 1989, pp. 364-369.

[4] K. M. CHANDY AND J. MISRA, *The drinking philosophers problem*, ACM Trans. Programming Languages and Systems, 6 (1984), pp. 632-646.

[5] M. CHOY AND A. K. SINGH, *Efficient fault tolerant algorithms for resource allocation in distributed systems*, in Proc. 24th ACM Symposium on Theory of Computing, 1992, pp. 593-602.

[6] R. COLE AND U. VISHKIN, *Deterministic coin tossing with applications to optimal parallel list ranking*, Inform. and Control, 70 (1986), pp. 32-53.

[7] G. N. FREDERICKSON AND N. A. LYNCH, *Electing a leader in a synchronous ring*, J. Assoc. Comput. Mach., 34 (1987), pp. 98-115.

[8] A. V. GOLDBERG, S. A. PLOTKIN, AND G. E. SHANNON, *Parallel symmetry-breaking in sparse graphs*, SIAM J. Discrete Math., 1 (1988), pp. 434-446.

[9] R. L. GRAHAM, B. L. ROTHSCHILD, AND J. H. SPENCER, *Ramsey Theory*, John Wiley, New York, 1980.

[10] N. LINIAL, *Locality in distributed graph algorithms*, SIAM J. Comput., 21 (1992), pp. 193-201.

[11] N. A. LYNCH, *Upper bounds for static resource allocation in a distributed system*, J. Comput. System Sci., 23 (1981), pp. 254-278.

[12] A. MAYER, M. NAOR, AND L. STOCKMEYER, *Local computations on static and dynamic graphs*, in Proc. 3rd Israel Symposium on Theory of Computing and Systems, Tel Aviv, Israel, 1995, pp. 268-278.

[13] S. MORAN, M. SNIR, AND U. MANBER, *Applications of Ramsey's theorem to decision tree complexity*, J. Assoc. Comput. Mach., 32 (1985), pp. 938-949.

[14] M. NAOR, *A lower bound on probabilistic algorithms for distributive ring coloring*, SIAM J. Discrete Math., 4 (1991), pp. 409-412.

[15] A. PANCONESI AND A. SRINIVASAN, *Improved distributed algorithms for coloring and network decomposition problems*, in Proc. 24th ACM Symposium on Theory of Computing, Victoria, BC, Canada, 1992, pp. 581-592.

[16] F. P. RAMSEY, *On a problem of formal logic*, Proc. London Math. Soc., (2), 30 (1930), pp. 264-286.

[17] E. STYER AND G. L. PETERSON, *Improved algorithms for distributed resource allocation*, in Proc. 7th ACM Symposium on Principles of Distributed Computing, Toronto, ON, Canada, 1988, pp. 105-116.

[18] A. C. YAO, *Should tables be sorted?*, J. Assoc. Comput. Mach., 28 (1981), pp. 615-628.

SIAM J. COMPUT.
Vol. 24, No. 6, pp. 1278-1304, December 1995

009

IDENTIFYING THE MINIMAL TRANSVERSALS OF A HYPERGRAPH AND RELATED PROBLEMS*

THOMAS EITER† AND GEORG GOTTLOB†

Abstract. The paper considers two decision problems on hypergraphs, hypergraph saturation and recognition of the transversal hypergraph, and discusses their significance for several search problems in applied computer science. Hypergraph saturation (i.e., given a hypergraph $\mathcal{H}$, decide if every subset of vertices is contained in or contains some edge of $\mathcal{H}$) is shown to be co-**NP**-complete. A certain subproblem of hypergraph saturation, the saturation of simple hypergraphs (i.e., Sperner families), is shown to be under polynomial transformation equivalent to transversal hypergraph recognition; i.e., given two hypergraphs $\mathcal{H}_1$, $\mathcal{H}_2$, decide if the sets in $\mathcal{H}_2$ are all the minimal transversals of $\mathcal{H}_1$. The complexity of the search problem related to the recognition of the transversal hypergraph, the computation of the transversal hypergraph, is an open problem. This task needs time exponential in the input size; it is unknown whether an output-polynomial algorithm exists. For several important subcases (for instance, if an upper or lower bound is imposed on the edge size or for acyclic hypergraphs) output-polynomial algorithms are presented. Computing or recognizing the minimal transversals of a hypergraph is a frequent problem in practice, which is pointed out by identifying important applications in database theory, Boolean switching theory, logic, and artificial intelligence (AI), particularly in model-based diagnosis.

Key words. hypergraphs, hypergraph transversals, hitting sets, independent sets, graph algorithms, NP-complete, polynomial time, output-polynomial time, database design, dependency inference, key computation, distributed databases, coteries, model-based diagnosis, satisfiability, prime implicants

AMS subject classifications. 05C65, 05C85, 05C90, 06E30, 68P15, 68Q20, 68Q25, 68R10, 68T30, 94C10

1. Introduction.

Hypergraph theory [7] is an important subfield of discrete mathematics with many relevant applications in both theoretical and applied computer science. In this paper, we study complexity issues of relevant computational problems on hypergraphs.

A hypergraph $\mathcal{H}$ is a set of subsets (edges) of a finite set of vertices. A hypergraph is simple if none of its edges is contained in any other of its edges. We say that a hypergraph is saturated if every subset of the vertex set is contained in an edge or contains an edge of the hypergraph.

The first problem we consider is the test of whether a given hypergraph $\mathcal{H}$ is saturated. This decision problem will be referred to as HYPERGRAPH SATURATION (H-SAT). We will also deal with a restricted version of this problem where $\mathcal{H}$ is supposed to be a simple hypergraph. This subproblem is called SIMPLE HYPERGRAPH SATURATION (SIMPLE-H-SAT).

The second main problem—and probably the more important one from the application viewpoint—concerns hypergraph transversals. A transversal (or hitting set) of a hypergraph $\mathcal{H}$ is a subset of the vertices of $\mathcal{H}$ which intersects each edge of $\mathcal{H}$. A transversal is minimal if it does not properly contain any other transversal. The set $Tr(\mathcal{H})$ of all minimal transversals of a hypergraph $\mathcal{H}$ is itself a hypergraph called the transversal hypergraph of $\mathcal{H}$.

Our second problem can thus be formulated as follows: Given two hypergraphs $\mathcal{G}$ and $\mathcal{H}$, decide whether $\mathcal{G}$ is the transversal hypergraph of $\mathcal{H}$. This decision problem, referred to as TRANSVERSAL HYPERGRAPH (TRANS-HYP), is closely related to the search problem of computing $Tr(\mathcal{H})$ for a given hypergraph $\mathcal{H}$.

The complexity of TRANS-HYP is currently an open issue. The problem is in co-**NP**, but there is no proof of co-**NP**-completeness. On the other hand, while the problem is polynomial-time solvable if $\mathcal{G}$ or $\mathcal{H}$ is a graph, no polynomial algorithm for the solution of the general version of TRANS-HYP is currently known. Similarly the question of whether $Tr(\mathcal{H})$ can be computed in output-polynomial total time (i.e., in time polynomial in the combined sizes of the

*Received by the editors June 3, 1993; accepted for publication (in revised form) July 7, 1994.

†Christian Doppler Laboratory for Expert Systems, Information Systems Department, Technical University of Vienna, Paniglgasse 16, A-1040 Wien, Austria ((eiter/gottlob)@vexpert.dbai.tuwien.ac.at).

input and the output) is an open problem. This complexity problem was posed independently by several researchers [41], [14], [29]. Note that the existence of an output-polynomial algorithm for computing $Tr(\mathcal{H})$ would imply the polynomial-time solvability of TRANS-HYP; on the other hand, if TRANS-HYP is co-**NP**-complete, then no output-polynomial algorithm for the computation of $Tr(\mathcal{H})$ is likely to exist.

This paper presents an extensive study of the aforementioned complexity and computation problems. We derive some new complexity results and exhibit relationships to other relevant problems such as satisfiability or hypergraph 2-colorability. Particular attention is paid to restricted versions of the general problem classes which are either polynomial-time solvable or $\leq_m^p$-equivalent (i.e., equivalent under polynomial-time transformations) to TRANS-HYP. The considered problems have important applications, e.g., in database theory, switching theory, and artificial intelligence (AI), from which it will become clear that computing the minimal transversals of a hypergraph is a frequently encountered problem of computer science. In the rest of this section, we outline the structure of the paper and highlight the most important results.

Our study of hypergraph problems starts in §2 with the definitions of basic hypergraph concepts and a brief overview of results known so far. In §3 we show that H-SAT is co-**NP**-complete and relate this problem to the well-known problem of hypergraph 2-colorability. SIMPLE-H-SAT turns out to be $\leq_m^p$-equivalent to the problem of 2-coloring a hypergraph whose edges are mutually intersecting. In §4 it is shown that SIMPLE-H-SAT is $\leq_m^p$-equivalent to TRANS-HYP. A restricted version of the latter problem is the question of whether a given hypergraph is equal to its own transversal hypergraph, i.e., $Tr(\mathcal{H}) = \mathcal{H}$. This problem, which we call SELFTRANSVERSALITY, has attracted much interest by mathematicians [36], [6], [7] but no complexity results have been derived so far. We show that SELF-TRANSVERSALITY is $\leq_m^p$-equivalent to TRANS-HYP.

Section 5 is dedicated to the investigation of polynomial subcases of the main problems. The most interesting results are briefly summarized as follows.

1. H-SAT becomes solvable in polynomial time if the cardinalities of the edges of the input hypergraph $\mathcal{H}$ differ by at most a constant k, in other words, if all edges of $\mathcal{H}$ have approximately the same size.

2. TRANS-HYP is decidable in polynomial time if the cardinalities of the edges of one of the input hypergraphs, say $\mathcal{H}$, are bounded by a constant. We present an output-polynomial algorithm for computing $Tr(\mathcal{H})$ in this case. Note that this result immediately leads to a relevant generalization of well-known output-polynomial methods for computing the maximal independent sets of graphs [52], [34], [29]. (Indeed, the maximal independent sets of a graph or hypergraph are exactly the complements of its minimal transversals.)

3. TRANS-HYP is decidable in polynomial time if one of the input hypergraphs is acyclic. We present an output-polynomial algorithm for computing $Tr(\mathcal{H})$ for acyclic hypergraphs $\mathcal{H}$. The type of acyclicity we consider is β-acyclicity as defined by Fagin in [16]. Note that β-acyclicity is among the weakest types of acyclicity; our results and methods also hold if we replace β-acyclicity by any of the several stronger types of acyclicity from the literature; cf. [16], [7].

Several applications, where recognizing or computing the transversal hypergraph plays a fundamental role, are described in §6. Our results can be profitably used in those applications. Section 7 concludes the paper; we state some open problems and give directions for further research.

2. Preliminaries and previous results.

DEFINITION 2.1. *A hypergraph is a pair* $(V, \mathcal{E})$ *of a finite set* $V = \{v_1, \ldots, v_n\}$ *and a set* $\mathcal{E}$ *of subsets of* V. *The elements of* V *are called vertices and the elements of* $\mathcal{E}$ *are called edges.*

Note that some authors, e.g. [7], state that the edge-set as well as each edge must be nonempty and that the union of all edges yields the vertex set.

For notational convenience, we will identify a hypergraph with its edge-set and vice versa if there is no danger of ambiguity. Thus, for hypergraphs $\mathcal{H} = (V, \mathcal{E})$ and $\mathcal{H}' = (V', \mathcal{E}')$, we write $E \in \mathcal{H}$ for $E \in \mathcal{E}$, $\mathcal{H} \cup \mathcal{H}'$ for $(V \cup V', \mathcal{E} \cup \mathcal{E}')$, $\min(\mathcal{H})$ for $(V, \min(\mathcal{E}))$, etc. Moreover, if not stated otherwise, it is presupposed that hypergraphs have the same set of vertices, which is denoted by V, n denotes the number of vertices, and m denotes the number of edges.

A hypergraph $\mathcal{H}$ is called *simple* if, for each pair H, H' of distinct edges of $\mathcal{H}$, it holds that $H \not\subseteq H'$. Simple hypergraphs are also called *Sperner families* according to [49], where it is proved that the cardinality $|\mathcal{H}|$ of a simple hypergraph $\mathcal{H}$ is bounded by $\binom{n}{\lfloor n/2 \rfloor}$, which is asymptotic to $\left(\frac{2}{\pi}\right)^{1/2} 2^n n^{-1/2}$ [3]. For example, the set of minimal models for a finite set of propositional clauses or the minimal keys of a database relation form a Sperner system.

Let $\mathcal{H}$ be a hypergraph. The set $V_e(\mathcal{H}) = \bigcup_{H \in \mathcal{H}} H$ are the *essential vertices* of $\mathcal{H}$. The *rank* $r(\mathcal{H})$ of $\mathcal{H}$ is defined by $r(\mathcal{H}) = \max\{|E| : E \in \mathcal{H}\}$, and its *antirank* $ar(\mathcal{H})$ is defined by $ar(\mathcal{H}) = \min\{|E| : E \in \mathcal{H}\}$. For example, every graph is a hypergraph of rank 2. $\mathcal{H}$ is called *intersecting* if no pair of its edges is disjoint, i.e., for all $E_1, E_2 \in \mathcal{H}$, $E_1 \cap E_2 \neq \emptyset$. Singleton edges in a hypergraph are referred to as *loops*.

Example. Consider the hypergraph $\mathcal{H} = \{\{1\}, \{2, 3\}, \{1, 3, 4\}\}$ on $V = \{1, 2, 3, 4\}$. We note that $V_e(\mathcal{H}) = V$, i.e., all vertices are essential, and $\mathcal{H}$ is not simple. Moreover, $ar(\mathcal{H}) = 1$, $r(\mathcal{H}) = 3$, and $\mathcal{H}$ is not intersecting.

DEFINITION 2.2. *Let $\mathcal{H}$ be a hypergraph. Then* $\min(\mathcal{H})$ *(resp.,* $\max(\mathcal{H})$*) denotes the set of minimal (resp., maximal) edges of $\mathcal{H}$ with respect to set inclusion, i.e.,* $\min(\mathcal{H}) = \{E \in \mathcal{H} :$ *there exists no $E' \in \mathcal{H}$ with $E' \subset E$*$\}$ *(resp.,* $\max(\mathcal{H}) = \{E \in \mathcal{H} :$ *there exists no $E' \in \mathcal{H}$ with $E' \supset E$*$\}$*).*

Clearly, $\min(\mathcal{H})$ and $\max(\mathcal{H})$ are simple hypergraphs on V. For the hypergraph $\mathcal{H} = \{\{1\}, \{2, 3\}, \{1, 3, 4\}\}$ of the last example, we have $\min(\mathcal{H}) = \{\{1\}, \{2, 3\}\}$ and $\max(\mathcal{H}) = \{\{2, 3\}, \{1, 3, 4\}\}$.

DEFINITION 2.3 (transversal). *Let $\mathcal{H}$ be a hypergraph. A set $T \subseteq V$ is a transversal of $\mathcal{H}$ if for each $E \in \mathcal{H}$, $T \cap E \neq \emptyset$. A transversal T is minimal if no proper subset T' of T is a transversal.*

Examples. Consider the hypergraph $\mathcal{H} = \{\{1\}, \{2, 3\}, \{1, 3, 4\}\}$ on $V = \{1, 2, 3, 4\}$ again. $\mathcal{H}$ has two minimal transversals: $\{1, 2\}$ and $\{1, 3\}$. Note that $\emptyset$ is a minimal transversal of $\mathcal{H} = \emptyset$, since every $T \subseteq V$ vacuously fulfills the transversal criterion, and $\mathcal{H} = \{\emptyset\}$ has no transversal, since no $T \subseteq V$ has a nonempty intersection with $\emptyset$.

Transversals are also called *hitting sets* and, in the case of a graph, *vertex covers*. The minimal vertex covers of a graph as well as the maximal independent sets are output-efficiently computable. An *independent set* of a graph $\mathcal{G}$ is a vertex subset $S \subseteq V$ that contains no edge of $\mathcal{G}$ and is maximal if no proper superset of it has this property. Clearly, S is a maximal independent set of $\mathcal{G}$ iff $V - S$ is a minimal vertex cover of $\mathcal{G}$. Finding the lexicographically first maximal independent set is LOGSPACE-hard [11], while finding the lexicographically last maximal independent set is **NP**-hard [29]. (Given that V is ordered, $V_1 \subseteq V$ comes lexicographically before $V_2 \subseteq V$ if the first element at which V_1 and V_2 disagree is in V_1 [29].) The maximal independent sets of a graph can be output with polynomial-time delay [52], [34], even if they have to be in lexicographic order [29]. Moreover, fast parallel algorithms are known; cf. [31].

Note that transversals are closely related to edge covers. An *edge cover* of a hypergraph $\mathcal{H}$ is an edge subset $\mathcal{H}' \subseteq \mathcal{H}$ such that $\bigcup_{E \in \mathcal{H}'} E$ yields V, and is minimal if no proper subset is an edge cover. The computation of minimal transversals and minimal edge covers is easily

transformable into each other (cf. [35]); thus results on one of these problems apply to the other.

Finding a minimal transversal of a hypergraph $\mathcal{H}$ is efficiently possible: if $\mathcal{H} = \emptyset$, then $\emptyset$ is the only minimal transversal of $\mathcal{H}$. If $\mathcal{H} \neq \emptyset$, then $V = \{v_1, \ldots, v_n\}$ is a transversal of $\mathcal{H}$. If we define

$$T_0 = V, \quad \text{and for all } i,\ 1 \leq i \leq n,$$

$$T_i = \begin{cases} T_{i-1} & \text{if } T_{i-1} - \{v_i\} \text{ is not a transversal,} \\ T_{i-1} - \{v_i\} & \text{if } T_{i-1} - \{v_i\} \text{ is a transversal,} \end{cases}$$

then T_n is a minimal transversal of $\mathcal{H}$. Thus a minimal transversal of $\mathcal{H}$ can be found in time $O(m \cdot n)$. Finding a minimum cardinality transversal, however, is **NP**-hard; cf. [21].

DEFINITION 2.4. *Let $\mathcal{H}$ be a hypergraph on V. The transversal hypergraph $Tr(\mathcal{H})$ of $\mathcal{H}$ is the hypergraph on V whose edges are the minimal transversals of $\mathcal{H}$.*

The following propositions capture important relations between a hypergraph $\mathcal{H}$ and $Tr(\mathcal{H})$; cf. [7].

PROPOSITION 2.1. *For every hypergraph $\mathcal{H}$, $Tr(\mathcal{H})$ is a simple hypergraph and $Tr(\mathcal{H}) = Tr(\min(\mathcal{H}))$.*

PROPOSITION 2.2. *Let $\mathcal{G}$ and $\mathcal{H}$ be simple hypergraphs. Then*

(i) *$\mathcal{G} = Tr(\mathcal{H})$ if and only if $\mathcal{H} = Tr(\mathcal{G})$;*

(ii) *$Tr(\mathcal{G}) = Tr(\mathcal{H})$ iff $\mathcal{G} = \mathcal{H}$;*

(iii) *$Tr(Tr(\mathcal{H})) = \mathcal{H}$.*

The following simple algorithm for determining $Tr(\mathcal{H})$ is given in [7] (cf. also [38], [35]): Let $\mathcal{H} = \{E_1, \ldots, E_m\}$, $m \geq 0$. Note that $Tr(\{E_i\}) = \{\{e\} : e \in E_i\}$. Define a sequence $\mathcal{T}_0, \ldots, \mathcal{T}_m$ by

$$\mathcal{T}_0 = \{\emptyset\}, \quad \text{and for all } i,\ 1 \leq i \leq m, \quad \mathcal{T}_i = \min\{T \cup \{e\} : T \in \mathcal{T}_{i-1} \text{ and } e \in E_i\}.$$

Then $\mathcal{T}_j = Tr(\{E_1, \ldots, E_j\})$, $0 \leq j \leq m$, and thus $\mathcal{T}_m = Tr(\mathcal{H})$.

It is easy to see that the computation of the transversal hypergraph is inherently exponential. For example, consider the hypergraph $\mathcal{F}_n = \{\{2i-1, 2i\} : 1 \leq i \leq n\}$ on $V_n = \{1, \ldots, 2n\}$. Then $Tr(\mathcal{F}_n) = \{\{x_1, \ldots, x_n\} : x_i \in \{2i, 2i-1\}, 1 \leq i \leq n\}$. Since $Tr(\mathcal{F}_n)$ has 2^n edges, the computation of $Tr(\mathcal{F}_n)$ needs space (hence also time) exponential in the input size.

Since an algorithm for computing $Tr(\mathcal{H})$ that runs in time polynomial in the input size (IS) is not possible, an algorithm is desirable which works at least in polynomial time if the number of minimal transversals (resp., the output size (OS)) is taken into account. There are various possibilities for defining appealing notions of output-polynomiality, e.g., output-polynomial total time, incremental-polynomial time, polynomial-time delay computability [29], or P-enumerability [54].

The most general of these concepts is *output-polynomial total time*, which requires that an algorithm work in time bounded by a polynomial in *IS and OS*. Note that an output-polynomial total time algorithm may run for exponentially many steps in IS without producing any output. The same holds for the notion of *P-enumerability*, where a search algorithm computes all solutions of a problem in time $p(IS)N$, where p is a polynomial and N is the number of solutions [54]. To consider incremental computations, an algorithm meets the *incremental-polynomial time* criterion if it outputs all solutions $s_1, \ldots, s_N$ to the respective problem one by one as follows: the time until the first output and the time between output of solutions s_i, s_{i+1}, $1 \leq i < N$, is bounded by a polynomial in the combined sizes of the input and all solutions $s_1, \ldots, s_i$, and the algorithm stops in polynomial time after the output of the

last solution. A more restrictive criterion is output with *polynomial-time delay*. An algorithm with this property generates the solutions one by one such that the time until the first solution is output and between the output of consecutive solutions is bounded by a polynomial in IS. Clearly, any such algorithm P-enumerates the solutions.

It is easy to show that the above algorithm does not stop in output-polynomial total time. Consider the complete graph $\mathcal{K}_n$ on vertices $\{1, \ldots, n\}$, i.e., $\mathcal{K}_n = \{\{i, j\} : 1 \leq i < j \leq n\}$, where $n \geq 4$ is even. Let the edges $E_1, \ldots, E_m$, $m = |\mathcal{K}_n|$, of $\mathcal{K}_n$ be ordered such that $E_1 = \{1, 2\}$, $E_2 = \{3, 4\}, \ldots, E_{n/2} = \{n-1, n\}$. Computing $Tr(\mathcal{K}_n)$ by $\mathcal{T}_0, \ldots, \mathcal{T}_m$, we find that $|\mathcal{T}_{n/2}| = 2^{n/2}$, because $\mathcal{T}_{n/2} = Tr(\{E_1, \ldots, E_{n/2}\})$ and $\{E_1, \ldots, E_{n/2}\}$ is just the hypergraph $\mathcal{F}_{n/2}$ from above. As $Tr(\mathcal{K}_n)$ is given by $\mathcal{T}_m = \{\{1, \ldots, n\} - \{i\} : 1 \leq i \leq n\}$ and $m = n(n-1)/2$, we have $n = |Tr(\mathcal{K}_n)| = \Theta(|\mathcal{K}_n|^{1/2})$. Relating $|\mathcal{T}_{n/2}|$ to the jth power, j arbitrarily fixed, of $|\mathcal{K}_n| = \max\{|\mathcal{K}_n|, |Tr(\mathcal{K}_n)|\}$ we get

$$|\mathcal{T}_{n/2}| = 2^{\Theta(|\mathcal{K}_n|^{1/2})}; \quad \text{thus} \quad |\mathcal{T}_{n/2}| > |\mathcal{K}_n|^j$$

for sufficiently large n. Hence, the size of $\mathcal{T}_{n/2}$ is not bounded by any polynomial in the input and output size. Consequently, the above algorithm is not efficient with respect to output-polynomiality.

Although there are several algorithms which involve, in a more or less obvious way, the computation of transversal hypergraphs, (e.g., [38], [35], [46], [14], [40], [42], [45]), unfortunately no output-polynomial transversal hypergraph algorithm is known today; moreover, it is even an open question whether such an algorithm is likely to exist at all; cf. [14], [29], [43]. We will show in §4 that the complexity of the closely related decision problem TRANS-HYP, which, given two hypergraphs $\mathcal{G}$ and $\mathcal{H}$, involves deciding if $\mathcal{G} = Tr(\mathcal{H})$, is of crucial importance to answer this question.

The second kind of problem in our study is the complexity of hypergraph saturation, a covering problem for the power set $\mathcal{P}(S)$ of a finite set S considered in [51], [10]. We introduce some notation first.

DEFINITION 2.5. *Let S be a finite set, and let $X \subseteq S$. Then $Cov_S(X) = \{Y \subseteq S : Y \subseteq X \text{ or } Y \supseteq X\}$. For any hypergraph $\mathcal{H}$ on V, $Cov(\mathcal{H}) = \bigcup_{H \in \mathcal{H}} Cov_V(H)$.*

In other words, $Cov(\mathcal{H})$ is the set of sets $X \subseteq V$ that are *covered* by an edge E of $\mathcal{H}$ *as subset* ($X \subseteq E$) or *as superset* ($X \supseteq E$).

DEFINITION 2.6. *A hypergraph $\mathcal{H}$ on V is called saturated if and only if $Cov(\mathcal{H}) = \mathcal{P}(V)$.*

According to the definition, a hypergraph $\mathcal{H}$ is saturated iff every set $V' \subseteq V$ is contained in at least one edge of $\mathcal{H}$ or contains at least one edge of $\mathcal{H}$.

Example. Consider the hypergraph $\mathcal{H} = \{\{1\}, \{2, 3\}, \{1, 3, 4\}\}$ on $V = \{1, 2, 3, 4\}$ again. Then $Cov(\mathcal{H}) = \mathcal{P}(V) - \{2, 4\}$, i.e., $\mathcal{H}$ covers all subsets of V but $\{2, 4\}$. Hence, $\mathcal{H}$ is not saturated. If we add $\{2, 4\}$ to $\mathcal{H}$, the resulting hypergraph $\mathcal{H}' = \{\{1\}, \{2, 3\}, \{1, 3, 4\}, \{2, 4\}\}$ is saturated. Note that saturation of a hypergraph $\mathcal{G}$ does not imply saturation of either $\min(\mathcal{G})$ or $\max(\mathcal{G})$. As an example consider $\mathcal{H}'$. Neither $\min(\mathcal{H}') = \{\{1\}, \{2, 3\}, \{2, 4\}\}$ nor $\max(\mathcal{H}') = \{\{2, 3\}, \{1, 3, 4\}, \{2, 4\}\}$ is saturated, because $\{3, 4\}$ and $\{1, 2\}$, respectively, are not covered.

The complexity of checking if a hypergraph is saturated has, to the best of our knowledge, not been studied yet in the literature.

We assume that the reader is familiar with the basic concepts of the theory of **NP**-completeness (cf. [21]). co-Π denotes the problem complementary to decision problem Π; $\leq_m^p$ denotes polynomial-time transformability. Decision problems Π and Π' are called $\leq_m^p$-*equivalent* iff $\Pi \leq_m^p \Pi'$ and $\Pi' \leq_m^p \Pi$. Problems that are solvable in polynomial time are referred to as *tractable*, and those which are most likely not solvable in polynomial time, among them all **NP**-hard problems, are referred to as *intractable*.

3. Hypergraph saturation. Imagine you are a friend of Toni, who manages a pizza restaurant that is famous for its rich variety of pizza.[1]

The pizzas differ in their toppings, and accordingly every pizza has its special name. For example, on a *pizza margherita* there are tomatoes, onions, and some cheese, on a *pizza al tonno* there is tuna fish and onions, and on a *pizza provinciale* there are tomatoes, ham, corn, onions, mushrooms, salami, pepperoni, and cheese. Now Toni wants to enlarge his pizza offer with a new pizza, but the new pizza should be neither the "grande" version of another pizza, i.e., have all the food of another pizza on it and some additional toppings, nor the "mini" version, i.e., all the food on it is also on some other pizza. Toni has been testing quite a number of food collections for the new pizza, but each of his compositions turned out to be a grande or a mini version of one of the many pizzas. This makes Toni wonder if there is any food collection for a new pizza at all. Since you are an expert in computer science, he asks you to write a computer program to answer this question. Under the principle of *quot capita tot gustus*, which means that every collection of food is appropriate for a pizza, you find that Toni's problem is just the hypergraph saturation problem if the food available is considered as set V of vertices, and the pizzas already offered, described by food collections, are considered as edges of a hypergraph. In a more precise formulation, we have the following

Problem. HYPERGRAPH SATURATION (H-SAT).

Instance. A hypergraph $\mathcal{H} = (V, \mathcal{E})$ on vertices $V = \{v_1, \ldots, v_n\}$.

Question. Is $\mathcal{H}$ saturated?

A simple brute force algorithm would be to test all possible 2^n vertex sets subsequently until some collection is found that is not covered by $\mathcal{H}$; this collection would be a new pizza which satisfies the requirements. However, if $\mathcal{H}$ is saturated, the algorithm recognizes this only after 2^n covering checks—too many if the number of food items on Toni's pizzas is taken into account, because then the algorithm would take weeks to run. Thus, an algorithm more subtle than the naive exponential algorithm is needed to solve the problem efficiently. However, chances are very low that there is some algorithm substantially faster than the brute force algorithm, because the problem is co-**NP**-complete; this is shown in this section. First, we need some additional definitions.

DEFINITION 3.1. *Let V be a set and let $V' \subseteq V$. Then $\overline{V'} = V - V'$. For every hypergraph $\mathcal{H}$ on V, the complemented hypergraph of $\mathcal{H}$ is the hypergraph $\overline{\mathcal{H}} = \{\overline{H} : H \in \mathcal{H}\}$ on V.*

The following relationships between $\mathcal{H}$ and $\overline{\mathcal{H}}$ are straightforward, but nevertheless important.

PROPOSITION 3.1. *Let $\mathcal{H}$ be a hypergraph. Then $\overline{\overline{\mathcal{H}}} = \mathcal{H}$ and $\mathcal{H}$ is simple iff $\overline{\mathcal{H}}$ is simple.*

DEFINITION 3.2. *A hypergraph $\mathcal{H}$ is called self-complemented if and only if $\mathcal{H} = \overline{\mathcal{H}}$.*

Note that if a hypergraph $\mathcal{H}$ is self-complemented, then for each $X \subseteq V$ we have $X \in Cov(\mathcal{H})$ *iff* $\overline{X} \in Cov(\mathcal{H})$.

2-colorability of a hypergraph, also known as set splitting, is defined as follows.

DEFINITION 3.3. *Let $\mathcal{H}$ be a hypergraph. A partitioning (A, B) of V, i.e., $A \cup B = V$ and $A \cap B = \emptyset$, is a 2-coloring of $\mathcal{H}$ iff for every $E \in \mathcal{H}$, it holds that $E \not\subseteq A$ and $E \not\subseteq B$. $\mathcal{H}$ is 2-colorable iff there exists a 2-coloring for $\mathcal{H}$.*

In other words, a hypergraph is 2-colorable if and only if there is an assignment of one of 2 "colors" to each vertex such that each edge has two colors. Deciding whether a hypergraph $\mathcal{H}$ is 2-colorable (HYPERGRAPH 2-COLORABILITY (HP2C)) is an **NP**-complete problem [36], [21]. We prove that this problem remains **NP**-complete for the following restriction.

THEOREM 3.2. HP2C *remains* **NP***-complete even if the hypergraph $\mathcal{H}$ is self-complemented.*

[1] Problems on pizza are studied elsewhere; cf. [24, p. 4].

Proof. Consider the hypergraph $\mathcal{H}' = (V', \mathcal{E}')$, where $V' = V \cup \{e, f\}$ for some $e, f \notin V$, and $\mathcal{E}' = \mathcal{H} \cup \{\overline{E} \cup \{e, f\} : E \in \mathcal{H}\}$. It is clear that $\mathcal{H}'$ can be constructed from $\mathcal{H}$ in polynomial time; moreover, $\mathcal{H}'$ is self-complemented. We claim that $\mathcal{H}$ is 2-colorable if and only if $\mathcal{H}'$ is 2-colorable. To prove the *only if* direction, let $(A, \overline{A})$ be any 2-coloring of $\mathcal{H}$. If e, f are colored differently, only an edge of $\mathcal{H}'$ that is also an edge of $\mathcal{H}$ can prevent a 2-coloring of $\mathcal{H}'$. Hence $(A \cup \{e\}, (V - A) \cup \{f\})$ is a 2-coloring of $\mathcal{H}'$. For the *if* direction, if $(B, \overline{B})$ is any 2-coloring of $\mathcal{H}'$, it does not color any edge from $\mathcal{H}$ monochromatically because $\mathcal{H} \subseteq \mathcal{H}'$, and since no edge of $\mathcal{H}$ contains e or f, clearly $(B - \{e, f\}, \overline{B} - \{e, f\})$ is a 2-coloring of $\mathcal{H}$. □

Using the relationship between 2-colorable hypergraphs and saturated hypergraphs, we now show the intractability of H-SAT. We refer to the following lemma.

LEMMA 3.3. *Let $\mathcal{H}$ be a self-complemented hypergraph. Then $\mathcal{H}$ is not 2-colorable iff $\mathcal{H}$ is saturated.*

Proof. If $\mathcal{H}$ is not 2-colorable, for every partitioning $(A, \overline{A})$ of V there exists $H \in \mathcal{H}$ such that $H \subseteq A$ or $H \subseteq \overline{A}$. Because self-complementarity of $\mathcal{H}$ implies that $A \in Cov(\mathcal{H})$ iff $\overline{A} \in Cov(\mathcal{H})$, it follows that $A, \overline{A} \in Cov(\mathcal{H})$; consequently, $\mathcal{H}$ is saturated, which proves the *only if* direction. For the *if* direction, assume $\mathcal{H}$ is 2-colored by $(A, \overline{A})$. We show that for every $H \in \mathcal{H}$, it holds that $A \not\supseteq H$ and $A \not\subseteq H$; hence, $A \notin Cov(\mathcal{H})$, which means that $\mathcal{H}$ is not saturated. Consider $H \in \mathcal{H}$. Since $(A, \overline{A})$ is a 2-coloring of $\mathcal{H}$, we have $A \not\supseteq H$. If $A \subseteq H$, then $\overline{A} \supseteq \overline{H}$, which by self-complementarity of $\mathcal{H}$ contradicts the fact that $(A, \overline{A})$ is a 2-coloring of $\mathcal{H}$. Thus $A \not\subseteq H$. This proves the *if* direction. □

THEOREM 3.4. H-SAT *is co-***NP***-complete, even if $\mathcal{H}$ is self-complemented.*

Proof. Membership in co-**NP** is easy to show: Guess a subset $V' \subseteq V$ and check if $V' \in Cov(\mathcal{H})$ holds. This can clearly be done in time polynomial in n and m. co-**NP**-hardness under the asserted restriction follows immediately from Theorem 3.2 and Lemma 3.3. □

This result can be strengthened as follows.

THEOREM 3.5. H-SAT *remains co-***NP***-complete even if $\mathcal{H}$ is self-complemented and all edges have size k or $n - k$ for fixed $k \geq 3$.*

Proof. HP2C remains **NP**-complete even if no edge of $\mathcal{H}$ contains more than three vertices [21]. Applying the transformation in the proof of Theorem 3.2 on such a hypergraph $\mathcal{H}$, we obtain a self-complemented hypergraph $\mathcal{H}' = (V', \mathcal{E}')$ such that every $E \in \mathcal{H}'$ contains at most three vertices or at least $n' - 3$ vertices, where $n' = |V'| = n + 2$, with the property that $\mathcal{H}'$ is saturated iff $\mathcal{H}$ is not 2-colorable. Hence, H-SAT remains co-**NP**-complete if $\mathcal{H}$ has the form of $\mathcal{H}'$. Therefore, in the following we assume without loss of generality that $\mathcal{H}$ is self-complemented and every edge contains at most three vertices or at least $n - 3$ vertices.

We first show the result for $k = 3$. Without loss of generality assume that $n > 2k = 6$. One can test in polynomial time if, for each $X \subseteq V$ with $|X| \leq 3$, $X \in Cov(\mathcal{H})$ holds. If $\mathcal{H}$ does not cover all the sets X, then $\mathcal{H}$ is not saturated. Otherwise, replace in $\mathcal{H}$ every edge E with $|E| < 3$ by all sets E' of size 3 such that $E \subset E' \subseteq V$ and every edge F with $|F| > n - 3$ by all subsets $F' \subset F$ of size $n - 3$. Let $\mathcal{H}''$ be the resulting hypergraph. Note that $\mathcal{H}''$ is self-complemented and contains only edges of size 3 or $n - 3$; furthermore, it follows from the construction of $\mathcal{H}''$ that $X \in Cov(\mathcal{H}'')$ for each $X \subseteq V$ such that $|X| \leq 3$.

We claim that $\mathcal{H}''$ is saturated iff $\mathcal{H}$ saturated. To show this, assume that $\mathcal{H}$ is not saturated. Then, since $\mathcal{H}$ is self-complemented, there exists $X \subseteq V$ such that $3 < |X| < n - 3$ and $X \notin Cov(\mathcal{H})$. By construction of $\mathcal{H}''$, it follows that $X \notin Cov(\mathcal{H}'')$. Thus $\mathcal{H}''$ is not saturated. The proof of the converse direction is analogous.

Since $\mathcal{H}''$ can be constructed in polynomial time, H-SAT remains co-**NP**-complete under the asserted restriction for $k = 3$.

For $k > 3$, the proof is analogous, where in each place "3" is replaced by "k." Note that since k is a constant, it can be checked in polynomial time if, for each $X \subseteq V$ with $|X| \leq k$, it holds that $X \in Cov(\mathcal{H})$ (there are fewer than kn^k such X), and $\mathcal{H}''$ can be constructed in polynomial time. □

The following result is an immediate consequence of Theorem 3.5 and Lemma 3.3.

COROLLARY 3.6. HP2C *is* **NP***-complete even if* $\mathcal{H}$ *is self-complemented and all edges have size* k *or* $n - k$ *for fixed* $k \geq 3$.

The case $k = 3$ in Theorem 3.5 marks the intractability frontier, as the case $k = 2$ is polynomial.

THEOREM 3.7. H-SAT *is polynomial if* $\mathcal{H}$ *is self-complemented and all edges have size* 2 *or* $n - 2$.

Proof. Let $\mathcal{G}$ be the graph on V with edges $\{E \in \mathcal{H} : |E| = 2\}$. We reduce deciding saturation of $\mathcal{H}$ to testing certain properties of $\mathcal{G}$, which can be done in polynomial time. In what follows, we assume without loss of generality that $\mathcal{G}$ is not empty (i.e., $\mathcal{H}$ is not empty) and $n \geq 4$.

CLAIM A. *Let* $v \in V$ *be arbitrary. If* v *belongs to all edges of* $\mathcal{G}$*, then* $\mathcal{H}$ *is not saturated iff* $V_e(\mathcal{G}) \subset V$.

To prove the *if* direction of this claim, assume that $V_e(\mathcal{G}) \subset V$. Let $v' \in V - V_e(\mathcal{G})$ be arbitrary. It is readily checked that the partitioning $(\{v, v'\}, V - \{v, v'\})$ is a 2-coloring of $\mathcal{H}$. Thus, by Lemma 3.3, $\mathcal{H}$ is not saturated. This proves the *if* direction. For the *only if* direction, assume that $\mathcal{H}$ is not saturated. Then, by Lemma 3.3, there must exist a 2-coloring of $\mathcal{H}$; let $(B, \overline{B})$ be such an arbitrary 2-coloring. Since $\mathcal{G} \subseteq \mathcal{H}$, $(B, \overline{B})$ is a 2-coloring of $\mathcal{G}$. Assume that $V_e(\mathcal{G}) = V$. Then it follows that $B = \{v\}$ or $\overline{B} = \{v\}$; assume without loss of generality that $B = \{v\}$. Let $E \in \mathcal{G}$ be arbitrary. Since $v \in E$, it follows that $\overline{E} \subseteq V - \{v\} = \overline{B}$. Since $\mathcal{H}$ is self-complemented, we have $\overline{E} \in \mathcal{H}$; this implies that $(B, \overline{B})$ is not a 2-coloring of $\mathcal{H}$, which is a contradiction. Thus it follows that $V_e(\mathcal{G}) \neq V$, and hence $V_e(\mathcal{G}) \subset V$.

CLAIM B. *If no* $v \in V$ *belongs to all edges of* $\mathcal{G}$*, then* $\mathcal{H}$ *is not saturated iff* $\mathcal{G}$ *is* 2*-colorable.*

The *only if* direction is immediate from Lemma 3.3. For the *if* direction, let $(B, \overline{B})$ be any 2-coloring of $\mathcal{G}$. Since no $v \in V$ belongs to all edges of $\mathcal{G}$, the 2-coloring must satisfy $|B \cap V_e(\mathcal{G})| \geq 2$ and $|\overline{B} \cap V_e(\mathcal{G})| \geq 2$. It follows that $B \notin Cov(\mathcal{G})$ and $\overline{B} \notin Cov(\mathcal{G})$. Since the latter is equivalent to $B \notin Cov(\overline{\mathcal{G}})$ and $\mathcal{H} = \mathcal{G} \cup \overline{\mathcal{G}}$, we have $B \notin Cov(\mathcal{H})$. This means that $\mathcal{H}$ is not saturated and the *if* direction is shown.

In summary, by Claims A and B, the problem of deciding whether $\mathcal{H}$ is saturated is efficiently reducible to checking whether $V_e(\mathcal{G}) \not\subset V$ or deciding whether $\mathcal{G}$ is not 2-colorable. The former is trivially polynomial, and the latter is polynomial since checking 2-colorability of a graph is polynomial [22].

It follows that H-SAT is polynomial under the asserted restriction. □

3.1. Simple hypergraph saturation. Toni has a brother Luigi, who also manages a big pizzeria. Luigi is proud that each of his pizzas is "originale," which means that in his restaurant there are no grande or mini versions of pizzas. As he hears from the plans of his brother, Luigi also intends to enlarge his pizza offer by a new pizza, which, of course, has to be a pizza originale. It is clear that Luigi's problem is a restricted version of Toni's problem. Since all pizzas are "originali," in the corresponding hypergraph saturation problem the hypergraph of pizzas is simple. Thus, in terms of hypergraphs, we have the following.

Problem. SIMPLE HYPERGRAPH SATURATION (SIMPLE-H-SAT).

Instance. A simple hypergraph $\mathcal{H} = (V, \mathcal{E})$ on vertices $V = \{v_1, \ldots, v_n\}$.

Question. Is $\mathcal{H}$ saturated?

The question is if this restriction on the input makes H-SAT tractable. Though SIMPLE-H-SAT is clearly in co-**NP**, a precise classification is still open.

A saturation test for two simple hypergraphs is easily reducible to a test for one hypergraph.

DEFINITION 3.4. *Let $\mathcal{H}$ be a hypergraph on V and let v be a (possibly new) vertex. The hypergraph $\mathcal{H}^v$ is defined by $\mathcal{H}^v = (V \cup \{v\}, \{H \cup \{v\} : H \in \mathcal{H}\})$. Let $\mathcal{G}$ and $\mathcal{H}$ be simple hypergraphs on V and let e, f be two (possibly new) vertices. Then $\mathcal{C}_{e,f}(\mathcal{G}, \mathcal{H}) = (V \cup \{e, f\}, \mathcal{G}^e \cup \mathcal{H}^f \cup \{V, \{e, f\}\})$.*

PROPOSITION 3.8. *Let $\mathcal{G}$ and $\mathcal{H}$ be simple hypergraphs which do not contain $\emptyset$ or V as edges and let e, $f \notin V$ be distinct. Then $\mathcal{C}_{e,f}(\mathcal{G}, \mathcal{H})$ is simple and $\mathcal{C}_{e,f}(\mathcal{G}, \mathcal{H})$ is saturated iff $\mathcal{G}$ and $\mathcal{H}$ are saturated.*

Proof. It is readily verified that $\mathcal{C}_{e,f}(\mathcal{G}, \mathcal{H})$ is simple. To show the other property, assume first that $\mathcal{C}_{e,f}(\mathcal{G}, \mathcal{H})$ is not saturated. That is, there exists $X \subseteq V \cup \{e, f\}$ such that $X \notin Cov(\mathcal{C}_{e,f}(\mathcal{G}, \mathcal{H}))$. Since $V \in \mathcal{C}_{e,f}(\mathcal{G}, \mathcal{H})$ and $\{e, f\} \in \mathcal{C}_{e,f}(\mathcal{G}, \mathcal{H})$, X must contain exactly one of e and f. If $e \in X$, it follows that $X - \{e\} \notin Cov(\mathcal{G})$; hence, $\mathcal{G}$ is not saturated. Analogously, if $f \in X$, it follows that $\mathcal{H}$ is not saturated. Thus the *if* direction holds. For the *only if* direction, assume without loss of generality that $\mathcal{G}$ is not saturated. Let $X \subseteq V$ such that $X \notin Cov(\mathcal{G})$. Then we have that $X \cup \{e\} \notin Cov(\mathcal{C}_{e,f}(\mathcal{G}, \mathcal{H}))$, i.e., $\mathcal{C}_{e,f}(\mathcal{G}, \mathcal{H})$ is not saturated. □

We obtain that, for the following subcase, SIMPLE-H-SAT remains as hard as in the general case.

PROPOSITION 3.9. SIMPLE-H-SAT *is $\leq_m^p$-equivalent to the subcase where $\mathcal{H}$ is self-complemented.*

Proof. Note that a hypergraph $\mathcal{H}$ is saturated if and only if $\overline{\mathcal{H}}$ is saturated: indeed, for every $X \subseteq V$, we have $X \in Cov(\mathcal{H})$ iff $\overline{X} \in Cov(\overline{\mathcal{H}})$. Since the hypergraph $\mathcal{C}_{e,f}(\mathcal{G}, \mathcal{H})$ is self-complemented for $\mathcal{H} = \overline{\mathcal{G}}$, the result follows from Proposition 3.8. □

In terms of colorability, the following result can be formulated as an immediate consequence of Proposition 3.9 and Lemma 3.3. Observe that HP2C for simple hypergraphs is still **NP**-hard, as a hypergraph $\mathcal{H}$ is 2-colorable iff $\min(\mathcal{H})$ is 2-colorable.

COROLLARY 3.10. HP2C *for simple and self-complemented $\mathcal{H}$ is $\leq_m^p$-equivalent to co-*SIMPLE-H-SAT.

4. Relating hypergraph transversals and saturation. Problems involving hypergraph transversals appear in many practical applications (cf. §6), so their computational complexity is of major interest. Moreover, several computational problems on hypergraphs can be reformulated in terms of transversals; for example, finding a 2-coloring for a hypergraph $\mathcal{H}$ is equivalent to finding a transversal T of $\mathcal{H}$ such that $\overline{T}$ is also a transversal.

In this section we first give a characterization of saturated hypergraphs in terms of a transversal relation, and then we will consider the problem of transversal recognition, where we will show that this problem is $\leq_m^p$-equivalent to SIMPLE-H-SAT. Moreover, we will investigate necessary and sufficient criteria for simple hypergraph saturation and transversal hypergraph recognition, and we will show how the latter problem reduces to a very specific subcase.

4.1. Definitions and basic facts. We need some additional definitions first.

DEFINITION 4.1 (cf. [7]). *Let $\mathcal{H}$ and $\mathcal{G}$ be two hypergraphs. Then $\mathcal{H} \geq \mathcal{G}$ iff for every $H \in \mathcal{H}$ there exists $G \in \mathcal{G}$ such that $H \supseteq G$,[2] and $\mathcal{H} \leq \mathcal{G}$ iff for every $H \in \mathcal{H}$ there exists $G \in \mathcal{G}$ such that $H \subseteq G$.*

For every hypergraph $\mathcal{H}$, (i) $\emptyset \geq \mathcal{H}$ and (ii) $\{\emptyset\} \geq \mathcal{H}$ iff $\emptyset \in \mathcal{H}$. Note that $\mathcal{H} \geq \mathcal{G}$ (resp., $\mathcal{H} \leq \mathcal{G}$) does not imply $\mathcal{G} \leq \mathcal{H}$ (resp., $\mathcal{G} \geq \mathcal{H}$); for example, let $\mathcal{H} = \{\{1\}\}$ and $\mathcal{G} = \{\{1\}, \{2\}\}$. $\leq$ is in some sense dual to $\geq$ with respect to complementation. For every hypergraph $\mathcal{H}$, $\mathcal{H} \leq \{V\}$ and $\{V\} \leq \mathcal{H}$ iff $V \in \mathcal{H}$. Note that $\geq$ and $\leq$ are reflexive, transitive, and define partial orders on the simple hypergraphs on V. We observe the following properties.

[2] In [7, p. 45] $\geq$ is denoted by $\prec$.

PROPOSITION 4.1. *Let $\mathcal{G}$ and $\mathcal{H}$ be hypergraphs. Then*

$$\mathcal{G} \geq \mathcal{H} \iff \mathcal{G} \geq \min(\mathcal{H}) \iff \min(\mathcal{G}) \geq \min(\mathcal{H}),$$
$$\mathcal{G} \leq \mathcal{H} \iff \mathcal{G} \leq \max(\mathcal{H}) \iff \max(\mathcal{G}) \leq \max(\mathcal{H}).$$

We have the following characterization of saturated hypergraphs.

THEOREM 4.2. *Let $\mathcal{H} \neq \emptyset$ be a hypergraph. Then the following statements* (i)-(iii) *are equivalent:*

(i) *$\mathcal{H}$ is saturated;* (ii) $Tr(\overline{\max(\mathcal{H})}) \geq \min(\mathcal{H})$; (iii) $Tr(\overline{\mathcal{H}}) \geq \mathcal{H}$.

Proof. (i) $\Rightarrow$ (ii). Assume $\mathcal{H}$ is saturated. Consider $T \in Tr(\overline{\max(\mathcal{H})})$. We have $T \in Cov(\mathcal{H})$, but since T is a transversal of $\overline{\max(\mathcal{H})}$, for every $E \in \mathcal{H}$ it holds that $T \not\subseteq E$. But then $E' \subseteq T$ for some $E' \in \mathcal{H}$, which implies $T \supseteq E''$ for some $E'' \in \min(\mathcal{H})$. It follows that $Tr(\overline{\max(\mathcal{H})}) \geq \min(\mathcal{H})$.

(ii) $\Rightarrow$ (iii). Check that $\overline{\max(\mathcal{H})} = \min(\overline{\mathcal{H}})$. Hence $Tr(\overline{\max(\mathcal{H})}) = Tr(\min(\overline{\mathcal{H}})) = Tr(\overline{\mathcal{H}})$. Since $Tr(\overline{\max(\mathcal{H})}) \geq \min(\mathcal{H})$, it follows from Proposition 4.1 that $Tr(\overline{\mathcal{H}}) \geq \mathcal{H}$.

(iii) $\Rightarrow$ (i). Assume $\mathcal{H}$ is not saturated, i.e., there exists $X \subseteq V$ such that $X \notin Cov(\mathcal{H})$. Since $X \not\subseteq E$ for every $E \in \mathcal{H}$, X is a transversal of $\overline{\mathcal{H}}$. Let $X' \subseteq X$ be a minimal transversal of $\overline{\mathcal{H}}$. Now consider any $E \in \mathcal{H}$; since $E \not\subseteq X$, we have $E \not\subseteq X'$. Consequently, $Tr(\overline{\mathcal{H}}) \not\geq \mathcal{H}$. □

COROLLARY 4.3. *To test if $Tr(\mathcal{H}) \geq \mathcal{H}$ holds for a hypergraph $\mathcal{H}$ is co-***NP***-complete even if $\mathcal{H}$ is self-complemented and contains only edges of size k and $n - k$ for fixed $k \geq 3$.*

Proof. Membership in co-**NP** follows since a guess for $T \in Tr(\mathcal{H})$, such that for every $E \in \mathcal{H}$, $T \not\supseteq E$, can be verified in polynomial time. co-**NP**-hardness under the asserted restriction follows from Theorems 4.2 and 3.5. □

We remark that the equivalence of (i) and (iii) in Theorem 4.2, restricted to simple hypergraphs, is implicit in a result of [51], which investigates Sperner families in the context of relational database theory. Note that this criterion does not describe the transversal hypergraph *explicitly*. We will provide such a description later in this section.

4.2. The main problem. Besides SIMPLE-H-SAT, the recognition of the transversal hypergraph is the second issue of central concern in our study. We will now show that the complexity of this decision problem is of importance in computing the transversal hypergraph, and then we will demonstrate $\leq_m^p$-equivalence of SIMPLE-H-SAT and transversal hypergraph recognition. Let us start with a precise problem statement.

Problem. TRANSVERSAL HYPERGRAPH (TRANS-HYP).

Instance. Two hypergraphs $\mathcal{G} = (V, \mathcal{E}_1)$ and $\mathcal{H} = (V, \mathcal{E}_2)$ on vertices $V = \{v_1, \ldots, v_n\}$.

Question. Does $\mathcal{H} = Tr(\mathcal{G})$ hold?

The problem is in co-**NP**, since a guess for $T \in Tr(\mathcal{G})$, such that $T \notin \mathcal{H}$, can be verified in polynomial time. However, there is no trivial evidence that the problem is in **NP**, and there is even less that it is in **P**. We observe that TRANS-HYP efficiently reduces to the following subproblem.

PROPOSITION 4.4. TRANS-HYP *is $\leq_m^p$-equivalent to the subcase in which $\mathcal{G}$ and $\mathcal{H}$ are simple.*

Proof. Note that $\mathcal{H} = Tr(\mathcal{G})$ implies that $\mathcal{H}$ is simple. Furthermore, $Tr(\mathcal{G}) = Tr(\min(\mathcal{G}))$. Since $\min(\mathcal{G})$ is a simple hypergraph and computable in polynomial time, the result follows. □

The significance of TRANS-HYP on the related search problem, namely, the computation of the transversal hypergraph, is captured by the following theorem.

THEOREM 4.5. *If* TRANS-HYP *is not in* **P**, *then no output-polynomial total time algorithm for computing the transversal hypergraph exists.*

Proof. Assume that an algorithm A computes $Tr(\mathcal{F})$ from $\mathcal{F}$ in output-polynomial total time. Let its run time be bounded by a polynomial $p(IS(\mathcal{F}), N)$, where IS denotes the input size and $N = |Tr(\mathcal{F})|$. To solve TRANS-HYP, construct an algorithm A' which works as follows: A' simulates A applied on $\mathcal{G}$, but does this for at most $p(IS(\mathcal{G}), |\mathcal{H}|)$ steps. If A did not terminate within $p(IS(\mathcal{G}), |\mathcal{H}|)$ steps, then $\mathcal{H} \neq Tr(\mathcal{G})$. Otherwise, A' compares the hypergraph output by A to $\mathcal{H}$; these hypergraphs are identical iff $\mathcal{H} = Tr(\mathcal{G})$. It is easy to see that the run time of A' can be bounded by a polynomial $q(IS(\mathcal{G}), |\mathcal{H}|)$. (The bookkeeping between simulation steps can be done in polynomial time.) Hence A' is polynomial with respect to the input size. The result follows. □

4.3. Characterizations of the transversal hypergraph. Since one can test in polynomial time for every $X \subseteq V$ whether X is a minimal transversal for a given hypergraph $\mathcal{H}$, we wonder if we can infer that $\mathcal{G} = Tr(\mathcal{H})$ if all edges of $\mathcal{G}$ are minimal transversals of $\mathcal{H}$ and vice versa, since this can be checked in polynomial time. The answer is no.

PROPOSITION 4.6. *Let $\mathcal{G}$ and $\mathcal{H}$ be simple hypergraphs. Then $\mathcal{H} \subseteq Tr(\mathcal{G})$ and $\mathcal{G} \subseteq Tr(\mathcal{H})$ does not imply $\mathcal{H} = Tr(\mathcal{G})$.*

Proof. We give a counterexample. Let $V = \{a, \ldots, g\}$. Consider the hypergraphs

$$\mathcal{H} = \{\{a,b,c\}, \{a,d,g\}, \{a,e,f\}, \{b,d,f\}, \{b,e,g\}, \{c,d,e\}, \{c,f,g\}, \{d,e,f,g\}, \{b,c,e,f\}, \{b,c,d,g\}, \{a,c,e,g\}, \{a,c,d,f\}, \{a,b,f,g\}\},$$

$$\mathcal{G} = \{\{a,b,c,d\}, \{a,b,c,e\}, \{a,b,c,f\}, \{a,b,c,g\}, \{a,b,d,g\}, \{a,c,d,g\}, \{a,d,e,g\}, \{a,d,f,g\}, \{a,b,e,f\}, \{a,c,e,f\}, \{a,d,e,f\}, \{a,e,f,g\}, \{a,b,d,f\}, \{b,c,d,f\}, \{b,d,e,f\}, \{b,d,f,g\}, \{a,b,e,g\}, \{b,c,e,g\}, \{b,d,e,g\}, \{b,e,f,g\}, \{a,c,d,e\}, \{b,c,d,e\}, \{c,d,e,f\}, \{c,d,e,g\}\}.$$

It holds that $\mathcal{H} \subseteq Tr(\mathcal{G})$ and $\mathcal{G} \subseteq Tr(\mathcal{H})$, but $\mathcal{G} \neq Tr(\mathcal{H})$ as $\{c,f,g\} \in Tr(\mathcal{H})$ and $\{c,f,g\} \notin \mathcal{G}$.[3] □

For further characterizations of the transversal hypergraph of a simple hypergraph, we introduce a special delete operator which removes from every edge of a hypergraph a vertex in all possible ways.

DEFINITION 4.2. *Let V be a finite set and let $Hyp(V) = 2^{2^V}$ denote the set of all hypergraphs on V. The mapping $\delta : Hyp(V) \to Hyp(V)$ is defined by $\delta(\mathcal{H}) = (V, \{H - \{v\} : H \in \mathcal{H}, v \in H\})$.*

Examples. For the hypergraph $\mathcal{H} = \{\{1\},\{2, 3\},\{1, 3, 4\}\}$ on $V = \{1, 2, 3, 4\}$, we have $\delta(\mathcal{H}) = \{\emptyset, \{2\}, \{3\}, \{1,3\}, \{1,4\}, \{3,4\}\}$. Note in particular that $\delta(\emptyset) = \emptyset$ and $\delta(\{\emptyset\}) = \emptyset$.

We observe the following relationships.

PROPOSITION 4.7. *Let $\mathcal{G}$ and $\mathcal{H}$ be simple hypergraphs such that $V_e(\mathcal{G}) \neq \emptyset$ and $V_e(\mathcal{H}) \neq \emptyset$. Then*

(i) *$\mathcal{H} \subseteq Tr(\mathcal{G})$ implies $\mathcal{H} \leq \overline{\delta(\mathcal{G})}$, but the converse does not hold;*

(ii) *$\overline{\delta(\mathcal{H})} \geq Tr(\mathcal{H})$, i.e., every edge of $\overline{\delta(\mathcal{H})}$ is a transversal of $\mathcal{H}$;*

(iii) *$Tr(\mathcal{H}) \leq \overline{\delta(\mathcal{H})}$, i.e., every minimal transversal of $\mathcal{H}$ is contained in some edge of $\overline{\delta(\mathcal{H})}$.*

---

[3] $\mathcal{H} = \mathcal{P}_7 \cup \overline{\mathcal{P}_7} - \{\{a,b,d,e\}\}$ and $\mathcal{G} = \{E \cup \{v\} : E \in \mathcal{P}_7 - \{\{c,f,g\}\}, v \in \overline{E}\}$ for the hypergraph $\mathcal{P}_7 = \{\{a,b,c\}, \{c,d,e\}, \{b,d,f\}, \{a,e,f\}, \{a,d,g\}, \{b,e,g\}, \{c,f,g\}\}$ on $V = \{a, \ldots, g\}$ from [7, p. 47, Fig. 1].

Proof. (i) Consider $T \in \mathcal{H}$. Since $T \in Tr(\mathcal{G})$, it follows that for every $v \in T$ there exists $G \in \mathcal{G}$ such that $(T - \{v\}) \cap G = \emptyset$; hence $T - \{v\} \subseteq \overline{G}$. This implies that $T \subseteq \overline{G} \cup \{v\} = \overline{G - \{v\}}$, from which it immediately follows that $\mathcal{H} \leq \overline{\delta(\mathcal{G})}$.

The following example shows that the converse does not hold. Consider $\mathcal{H} = \{\{1\}\}$ and $\mathcal{G} = \{\{1\}, \{2\}\}$ on $V = \{1, 2\}$. Then $\overline{\delta(\mathcal{G})} = \{\{1, 2\}\}$ and $\mathcal{H} \leq \overline{\delta(\mathcal{G})}$, but $\mathcal{H} \not\subseteq Tr(\mathcal{G}) = \{\{1, 2\}\}$.

(ii) Consider $H \in \mathcal{H}$. $\overline{H}$ is not a transversal of $\mathcal{H}$, but since $\mathcal{H}$ is simple, it is a transversal of the hypergraph $\mathcal{H} - \{H\}$. Thus, for every $v \in H$, $\overline{H} \cup \{v\}$ is a transversal of $\mathcal{H}$; hence $\overline{H} \cup \{v\} = \overline{H - \{v\}}$ contains a minimal transversal of $\mathcal{H}$. The claim follows immediately from the definition of δ.

(iii) For every minimal transversal T of $\mathcal{H}$ and every vertex $v \in T$ there exists an edge H of $\mathcal{H}$ such that $T \cap H = \{v\}$; hence $T \subseteq \overline{H} \cup \{v\}$, but $\overline{H} \cup \{v\} = \overline{H - \{v\}} \in \overline{\delta(\mathcal{H})}$. □

Note that in the above proposition (i) states a necessary but not sufficient condition for TRANS-HYP. With the δ-operator we have the following important characterization of saturated simple hypergraphs.

THEOREM 4.8. *Let $\mathcal{H}$ be a simple hypergraph. Then $\mathcal{H}$ is saturated if and only if* $Tr(\mathcal{H}) = \min(\overline{\delta(\mathcal{H})})$.

Proof. We prove the *only if* direction first. Assume $\mathcal{H}$ is saturated, but $Tr(\mathcal{H}) \neq \min(\overline{\delta(\mathcal{H})})$. Note that this assumption implies $\mathcal{H} \neq \emptyset$ and $\mathcal{H} \neq \{\emptyset\}$, hence $V_e(\mathcal{H}) \neq \emptyset$.

CLAIM C. $Tr(\mathcal{H}) \not\subseteq \min(\overline{\delta(\mathcal{H})})$.

Proof. Suppose to the contrary that $Tr(\mathcal{H}) \subseteq \min(\overline{\delta(\mathcal{H})})$; hence, since $Tr(\mathcal{H}) \neq \min(\overline{\delta(\mathcal{H})})$, there exists an edge $E \in \min(\overline{\delta(\mathcal{H})}) - Tr(\mathcal{H})$. From Proposition 4.7(ii) it follows that E is a transversal of $\mathcal{H}$; thus E contains some $T \in Tr(\mathcal{H})$. Consider any such T. Since $\min(\overline{\delta(\mathcal{H})})$ is simple, it follows that $T \notin \min(\overline{\delta(\mathcal{H})})$. Hence T gives rise to a contradiction of $Tr(\mathcal{H}) \subseteq \min(\overline{\delta(\mathcal{H})})$. This proves the claim.

From Claim C and the assumption $Tr(\mathcal{H}) \neq \min(\overline{\delta(\mathcal{H})})$, it follows that there exists $T \in Tr(\mathcal{H})$ such that $T \in Tr(\mathcal{H}) - \min(\overline{\delta(\mathcal{H})})$.

CLAIM D. $\overline{H} \cap \overline{T} \neq \emptyset$.

Proof. Assume to the contrary that $\overline{H} \cap \overline{T} = \emptyset$, i.e., $\overline{H} \subseteq T$. Clearly, $\overline{H}$ is not a transversal of $\mathcal{H}$; however, since $\mathcal{H}$ is simple, for every $v \in H$ it holds that $\overline{H} \cup \{v\}$ is a transversal of $\mathcal{H}$. From this and $\overline{H} \subseteq T$, $T \in Tr(\mathcal{H})$ we infer that $T = \overline{H} \cup \{v'\}$ for some $v' \in H$. Hence $T \in \overline{\delta(\mathcal{H})}$, which means that T is a superset of some edge in $\min(\overline{\delta(\mathcal{H})})$. Since $T \in Tr(\mathcal{H})$, we infer from Proposition 4.7(ii) that $T \in \min(\overline{\delta(\mathcal{H})})$. We have reached a contradiction. Thus $\overline{H} \not\subseteq T$, i.e., $\overline{H} \cap \overline{T} \neq \emptyset$.

Since for each $H \in \mathcal{H}$, a vertex e exists such that $E = \overline{H} \cup \{e\} \in \overline{\delta(\mathcal{H})}$, it follows from Claim D that $\overline{T}$ is a transversal of $\overline{\mathcal{H}}$. Since $\mathcal{H}$ is saturated, we have by Theorem 4.2(iii) that $Tr(\overline{\mathcal{H}}) \geq \mathcal{H}$. Thus there exists $H \in \mathcal{H}$ such that $\overline{T} \supseteq H$, which is equivalent to $T \subseteq \overline{H}$. But this means that T is not a transversal of $\mathcal{H}$; this is a contradiction to the existence of T which followed from Claim C. Thus, the assumption $Tr(\mathcal{H}) \neq \min(\overline{\delta(\mathcal{H})})$, from which we derived Claim C, is not consistent. This proves the *only if* direction.

To prove the *if* direction, assume that $Tr(\mathcal{H}) = \min(\overline{\delta(\mathcal{H})})$, but $\mathcal{H}$ is not saturated. Note that $\mathcal{H} \neq \emptyset$. Since $\mathcal{H}$ is not saturated, there exists a set $T \subseteq V$ such that $T \notin Cov(\mathcal{H})$. In particular, we have for every $H \in \mathcal{H}$ that $T \not\supseteq H$, and hence $\overline{T} \cap H \neq \emptyset$. This means that $\overline{T}$ is a transversal of $\mathcal{H}$. From the assumption $Tr(\mathcal{H}) = \min(\overline{\delta(\mathcal{H})})$, we infer that $\overline{T} \supseteq E$ for some $E \in \min(\overline{\delta(\mathcal{H})})$. By the definition of $\delta(\mathcal{H})$, we have that $E = \overline{H} \cup \{v\}$ for some H and v such that $H \in \mathcal{H}$ and $v \in H$. Hence $\overline{T} \supseteq \overline{H} \cup \{v\}$, and thus $\overline{T} \supseteq \overline{H}$. The latter means $T \subseteq H$, which implies $T \in Cov(\mathcal{H})$. Thus we have reached a contradiction of the existence of T. This proves the *if* direction. □

Given a simple hypergraph $\mathcal{H}$, one can easily compute $\mathcal{H}' = \min(\overline{\delta(\mathcal{H})})$ in polynomial time. Since checking whether each edge of $\mathcal{H}$ is a minimal transversal of $\mathcal{H}'$ and vice versa is possible in polynomial time, we wonder if this necessary condition for saturation of $\mathcal{H}$ is also sufficient. Unfortunately, this does not hold.

PROPOSITION 4.9. *Let $\mathcal{H}$ be a simple hypergraph and let $\mathcal{H}' = \min(\overline{\delta(\mathcal{H})})$. Then $\mathcal{H} \subseteq Tr(\mathcal{H}')$ and $\mathcal{H}' \subseteq Tr(\mathcal{H})$ does not imply that $\mathcal{H}$ is saturated. The same holds if $\mathcal{H}$ is self-complemented.*

A counterexample for unrestricted $\mathcal{H}$ is not hard to find (e.g., in the proof of Proposition 4.6, $\mathcal{G} = \min(\overline{\delta(\mathcal{H})})$). Imposing self-complementarity on $\mathcal{H}$ makes this more challenging, in particular, to find a counterexample with as few edges or vertices as possible; we can offer an instance of $\mathcal{H}$ with 308 edges on 22 vertices.

It follows immediately from Theorem 4.8 that SIMPLE-H-SAT is $\leq_m^p$-transformable into TRANS-HYP. The next theorem implies the converse, which establishes that SIMPLE-H-SAT and TRANS-HYP are $\leq_m^p$-equivalent. We use the following simple but helpful lemma.

LEMMA 4.10. *If $\mathcal{H}$ is simple, then for each $E \in Tr(\mathcal{H})$, there exists no $E' \in \mathcal{H}$ such that $E' \subseteq \overline{E}$.*

Proof. Indeed, if $E \in Tr(\mathcal{H})$ then $E \cap E' \neq \emptyset$ for all $E' \in \mathcal{H}$; hence $E' - \overline{E} \neq \emptyset$, i.e., $E' \not\subseteq \overline{E}$. □

THEOREM 4.11. *Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two simple hypergraphs on V and let $e, f \notin V$ be distinct. Define a hypergraph $\mathcal{H} = (V', \mathcal{F}_1 \cup \mathcal{F}_2)$, where $V' = V \cup \{e, f\}$, $\mathcal{F}_1 = \mathcal{G}_1$, and $\mathcal{F}_2 = \{\overline{E} \cup \{e, f\} : E \in \mathcal{G}_2\}$. Then $\mathcal{G}_1 = Tr(\mathcal{G}_2)$ iff $\mathcal{H}$ is simple and saturated.*

Proof. For the *only if* direction, let $\mathcal{G}_1 = Tr(\mathcal{G}_2)$ hold. We first show that $\mathcal{H}$ is simple. Suppose $\mathcal{H}$ is not simple. Since $\mathcal{F}_1, \mathcal{F}_2$ constitute simple hypergraphs, there must exist $F_1 \in \mathcal{F}_1, F_2 \in \mathcal{F}_2$ such that $F_1 \subseteq F_2$ or $F_2 \subseteq F_1$. By Lemma 4.10 and the construction of $\mathcal{F}_1$ and $\mathcal{F}_2$ it follows that $F_1 \subseteq F_2$ is impossible to hold. However, $F_2 \subseteq F_1$ is also impossible because $e \in F_2$ but $e \notin F_1$. Thus we have reached a contradiction, and $\mathcal{H}$ is simple. Now we show that $\mathcal{H}$ must also be saturated. Assume this is not the case, i.e., there exists $X \subseteq V', X \notin Cov(\mathcal{H})$. Since for all $F \in \mathcal{F}_2$ we have $X \not\subseteq F$, it follows that X is a transversal of $\overline{\mathcal{F}_2} = \overline{\overline{\mathcal{G}_2}} = \mathcal{G}_2$. Since $\mathcal{F}_1$ constitutes $Tr(\mathcal{G}_2)$, this implies that $E \subseteq X$ for some $E \in \mathcal{F}_1$, which means that $X \in Cov(\mathcal{H})$, a contradiction. Thus $\mathcal{H}$ is saturated; the *only if* direction is proved.

To prove the *if* direction, assume that $\mathcal{H}$ is simple and saturated, but $\mathcal{G}_1 \neq Tr(\mathcal{G}_2)$. We show that $\mathcal{G}_1 \not\supset Tr(\mathcal{G}_2)$. Assume that $\mathcal{G}_1 \supset Tr(\mathcal{G}_2)$. Hence there exists a set $E \in \mathcal{G}_1 - Tr(\mathcal{G}_2)$. Since $\mathcal{G}_1 \supset Tr(\mathcal{G}_2)$ and $\mathcal{G}_1$ is simple, it follows that E is not a transversal of $\mathcal{G}_2$. Since E is not a transversal of $\mathcal{G}_2$, there exists some $E' \in \mathcal{G}_2$ such that $E \cap E' = \emptyset$; hence $E \subset \overline{E'} \cup \{e, f\}$. By construction of $\mathcal{H}$, we have that $E, \overline{E'} \cup \{e, f\} \in \mathcal{H}$. This means that $\mathcal{H}$ is not simple, which is a contradiction. Thus it follows that $\mathcal{G}_1 \not\supset Tr(\mathcal{G}_2)$.

Since $\mathcal{G}_1 \neq Tr(\mathcal{G}_2)$ and $\mathcal{G}_1 \not\supset Tr(\mathcal{G}_2)$, there exists a $T \in Tr(\mathcal{G}_2) - \mathcal{G}_1$. Consider $T \cup \{e\}$. Because T is a transversal of $\mathcal{G}_2 = \overline{\mathcal{F}_2}$, it is clear that there is no $F \in \mathcal{F}_2$ such that $T \cup \{e\} \subseteq F$. There is no $F \in \mathcal{F}_2$ such that $T \cup \{e\} \supseteq F$ because $f \in F$, $f \notin T$, and there is no $F \in \mathcal{F}_1$ with the property $T \cup \{e\} \subseteq F$ because $e \notin F$. Since $\mathcal{H}$ is saturated, this implies that there is some $F \in \mathcal{F}_1$ such that $F \subseteq T \cup \{e\}$, from which $F \subseteq T$ clearly follows. Because $\mathcal{H}$ is simple and $\mathcal{F}_2 \subseteq \mathcal{H}$, it holds for every $G \in \mathcal{F}_2$ that $F \not\subseteq G$; thus F is a transversal of $\overline{\mathcal{F}_2} = \mathcal{G}_2$. Since $T \in Tr(\mathcal{G}_2)$, it follows that $F = T$. Hence $T \in \mathcal{G}_1$, which is a contradiction of $T \in Tr(\mathcal{G}_2) - \mathcal{G}_1$. This proves the *if* direction. □

Now the aforementioned relationship between SIMPLE-H-SAT and TRANS-HYP is easy to establish.

THEOREM 4.12. SIMPLE-H-SAT *is $\leq_m^p$-equivalent to* TRANS-HYP.

Proof. Theorems 4.8 and 4.11 imply that SIMPLE-H-SAT $\leq_m^p$-reduces to TRANS-HYP and vice versa. □

4.4. Self-transversal hypergraphs. As previously mentioned, we show that recognizing the transversal hypergraph $\leq_m^p$-reduces to the SELF-TRANSVERSALITY problem, which is as follows.

Problem. SELF-TRANSVERSALITY.

Instance. A hypergraph $\mathcal{H} = (V, \mathcal{E})$ on vertices $V = \{v_1, \ldots, v_n\}$.

Question. Does $\mathcal{H} = Tr(\mathcal{H})$ hold?

Self-transversal hypergraphs have attracted much interest by mathematicians; cf. [36], [6], [7]. The simplest example of such a hypergraph is $\mathcal{H} = (\{x\}, \{\{x\}\})$. Another example is the hypergraph $\mathcal{H} = \{\{x, y\}, \{x, y\}, \{y, z\}\}$. For a thorough study of self-transversal hypergraphs, the reader is referred to [7, Chap. 2]. We have the following result.

THEOREM 4.13. *Deciding whether a hypergraph $\mathcal{H}$ satisfies $Tr(\mathcal{H}) = \mathcal{H}$ is $\leq_m^p$-equivalent to* SIMPLE-H-SAT.

Proof. We show that TRANS-HYP is $\leq_m^p$-reducible to SELF-TRANSVERSALITY, from which, by Theorem 4.12, the result follows. Without loss of generality, let $\mathcal{G}$ and $\mathcal{H}$ be two simple, nonempty hypergraphs on V and let $e, f \notin V$ be two new distinct vertices. Define a hypergraph $\mathcal{D}$ on $V' = V \cup \{e, f\}$ by $\mathcal{D} = \mathcal{G}^e \cup \mathcal{H}^f \cup \{\{e, f\}\}$. We claim that $\mathcal{G} = Tr(\mathcal{H})$ iff $\mathcal{D} = Tr(\mathcal{D})$.

To prove this, consider $Tr(D)$. We have that $\{e, f\} \in Tr(\mathcal{D})$ and also $\{e, f\} \in \mathcal{D}$. Every $T \in Tr(\mathcal{D}) - \{\{e, f\}\}$ distinct from $\{e, f\}$ contains either e or f, but not both. Assume that T contains e but not f. Since no edge in $\mathcal{H}^f$ contains e and all other edges in $\mathcal{D} - \mathcal{H}^f$ contain e, we infer that $T - \{e\}$ must be a transversal of $\mathcal{H}$. Conversely, if T' is a transversal of $\mathcal{H}$, then $T' \cup \{e\}$ is easily shown to be a transversal of $\mathcal{D}$. This implies that the minimal transversals of $\mathcal{D}$ which contain e but not f are given by $\mathcal{T}_e = \{E \cup \{e\} : E \in Tr(\mathcal{H})\} = Tr(\mathcal{H})^e$. From this it is immediately verified that $\mathcal{G} = Tr(\mathcal{H})$ if and only if $\mathcal{G}^e = \mathcal{T}_e$. In the same way, one can show that the set $\mathcal{T}_f$ of minimal transversals of $\mathcal{D}$ which contain f but not e satisfies $\mathcal{T}_f = \mathcal{H}^f$ if and only if $\mathcal{H} = Tr(\mathcal{G})$. Since $\mathcal{G}$ and $\mathcal{H}$ are simple, by Proposition 2.2, $\mathcal{G} = Tr(\mathcal{H})$ iff $\mathcal{H} = Tr(\mathcal{G})$. Hence, since $Tr(\mathcal{D}) = \mathcal{T}_e \cup \mathcal{T}_f \cup \{\{e, f\}\}$, we have that $\mathcal{G} = Tr(\mathcal{H})$ iff $\mathcal{D} = Tr(\mathcal{D})$; the claim is proved. □

From this result, by the close relation between 2-colorable intersecting hypergraphs and self-transversal hypergraphs [7], we obtain the following result on deciding 2-colorability of intersecting hypergraphs.

COROLLARY 4.14. HP2C *for intersecting hypergraphs is $\leq_m^p$-equivalent to co-*SIMPLE-H-SAT.

Proof. Note that $\mathcal{H}$ is intersecting iff $\min(\mathcal{H})$ is intersecting and $\mathcal{H}$ is 2-colorable iff $\min(\mathcal{H})$ is 2-colorable. Thus, without loss of generality we may assume that $\mathcal{H}$ is simple and intersecting and, in addition, $\mathcal{H}$ contains no loops, i.e., singleton edges. A simple intersecting hypergraph $\mathcal{H}$ without loops satisfies $\mathcal{H} = Tr(\mathcal{H})$ if and only if it is not 2-colorable [7]. Thus, if $\mathcal{H}$ is intersecting, 2-colorability of $\mathcal{H}$ is $\leq_m^p$-reducible to co-SELF-TRANSVERSALITY, which, by Theorem 4.13, is $\leq_m^p$-equivalent to co-SIMPLE-H-SAT. □

5. Polynomial cases. Although we do not know whether SIMPLE-H-SAT and TRANS-HYP are intractable in their general problem statement, in practical occurrences there are (natural) restrictions on the instances of a problem such that the problem reduces to a subproblem of the general problem. Since a subproblem might be computationally easier, there is hope to get efficient algorithms for it even if the more general problem is intractable. For example, in the pizza baker's problem it seems quite reasonable to assume that all pizzas have approximately the same number of food items; in this case, however, the problem becomes polynomial-time solvable for both Toni and Luigi. Narrowing the "frontier" between the in-

tractable general problem and tractable subcases [21], we identify some important subcases of SIMPLE-H-SAT and TRANS-HYP which are in **P**, among them the recognition of the minimal transversals of a hypergraph with bounded edge size and hypergraph saturation if the difference between rank and antirank is bounded. In particular, we present algorithms for output-efficient computation of the transversal hypergraph if the edge-size is small (probably the most important of the restrictions treated for practice) and if it is large, or if the hypergraph is acyclic.

5.1. Restrictions on the edge size. Let us consider restrictions on the size of the edges of a hypergraph. A straightforward restriction of this kind which is important in practice is a constant upper bound. Fortunately, the transversal hypergraph is output-efficiently computable in this case, which implies that the minimal transversals of such a hypergraph can be recognized in polynomial time and also saturation of a simple hypergraph of this form is efficiently decidable.

DEFINITION 5.1. *Let $\mathcal{H}$ be a hypergraph on vertices V, and let $x \in V$. The degree $d(x, \mathcal{H})$ of vertex x in hypergraph $\mathcal{H}$ is defined as $d(x, \mathcal{H}) = |\{E \in \mathcal{H} : x \in E\}|$.*

In the proof we apply a result from combinatorial studies by Berge and Duchet.

LEMMA 5.1 ([7, p. 58, Cor. 1]). *Let $\mathcal{G}$ be a simple hypergraph and let $k \geq 2$ be an integer. Then $r(Tr(\mathcal{G})) \leq k$ iff for all $\mathcal{G}' \subseteq \mathcal{G}$ with $|\mathcal{G}'| = k + 1$ there exists an $E \in \mathcal{G}$ subject to (s.t.) $E \subseteq \{v \in V : d(v, \mathcal{G}') > 1\}$.*

THEOREM 5.2. *Let $\mathcal{H}$ be a hypergraph. If the size of the edges of $\mathcal{H}$ is bounded by some constant k, i.e., $r(\mathcal{H}) \leq k$, then $Tr(\mathcal{H})$ is computable in incremental-polynomial time.*

Proof. We describe an algorithm which has the desired property. The following characterization of $Tr(\mathcal{H})$ for a simple, nonempty hypergraph $\mathcal{H}$ is obtained from Lemma 5.1. Let $r = r(\mathcal{H})$ and, for any hypergraph $\mathcal{F}$ and integer $i \geq 0$, denote by $\mathcal{F}|_i$ the hypergraph $\mathcal{F}_i = \{E \in \mathcal{F} : |E| \leq i\}$. Then

$$\mathcal{G} = Tr(\mathcal{H}) \iff (\mathcal{G} \subseteq Tr(\mathcal{H})) \wedge \mathbf{P} \wedge \mathbf{Q}, \tag{1}$$

where

$$\mathbf{P} \equiv Tr(\mathcal{G})|_r \subseteq \mathcal{H},$$

$$\mathbf{Q} \equiv \text{there exists no } \mathcal{G}' \subseteq \mathcal{G} \text{ with } |\mathcal{G}'| = r + 1 \text{ such that for all } E \in \mathcal{G},$$
$$E \not\subseteq \{x \in V : d(x, \mathcal{G}') > 1\}.$$

It is readily checked that (1) holds if $r \leq 1$. We show that (1) also holds if $r \geq 2$. If $\mathcal{G} = Tr(\mathcal{H})$ and $\mathcal{H}$ is simple, then $\mathcal{H} = Tr(\mathcal{G})$ and the *only if* direction follows by Lemma 5.1. For the *if* direction, if $\mathcal{H}$ is simple, $\mathcal{G} \subseteq Tr(\mathcal{H})$, and **P** holds, then $\mathcal{H} \subseteq Tr(\mathcal{G})$. To show this, consider $H \in \mathcal{H}$. Since $\mathcal{G} \subseteq Tr(\mathcal{H})$, H is a transversal of $\mathcal{G}$. Let $H' \subseteq H$ be an arbitrary minimal transversal of $\mathcal{G}$. Since $|H'| \leq r$, it follows from **P** that $H' \in \mathcal{H}$; as $\mathcal{H}$ is simple, it follows that $H' = H$. Hence $\mathcal{H} \subseteq Tr(\mathcal{G})$.

Since, by validity of **Q** and Lemma 5.1, $r(Tr(\mathcal{G})) \leq r$, it follows from **P** that $\mathcal{H} = Tr(\mathcal{G})$.

We now derive from (1) a method for incrementally computing $Tr(\mathcal{H})$, where $\mathcal{H}$ is simple and nonempty. If $\mathcal{G} \subseteq Tr(\mathcal{H})$ and $\mathcal{G} \neq Tr(\mathcal{H})$, then **P** or **Q** is false. In this case some $T_1 \in Tr(\mathcal{H})$, such that $T_1 \notin \mathcal{G}$, can be found as follows.

(i) Assume that **P** is false. Hence there exists $T \in Tr(\mathcal{G})|_r - \mathcal{H}$.

CLAIM E. *$\overline{T}$ is a transversal of $\mathcal{H}$.*

Proof. Note that $\overline{T}$ is not a transversal of $\mathcal{H}$ iff there exists an $H \in \mathcal{H}$ such that $\overline{T} \cap H = \emptyset$, which is equivalent to $H \subseteq T$. Since $\mathcal{G} \subseteq Tr(\mathcal{H})$, H is a transversal of $\mathcal{G}$. As $T \in Tr(\mathcal{G})|_r - \mathcal{H}$, T is a minimal transversal of $\mathcal{G}$. Since $H \subseteq T$, it follows that $H = T$; hence $T \in \mathcal{H}$. This is a contradiction. Consequently, $\overline{T}$ is a transversal of $\mathcal{H}$.

Algorithm Trans($\mathcal{H}$)
input: simple, nonempty hypergraph $\mathcal{H}$ on V.
output: $Tr(\mathcal{H})$ edge by edge, i.e., incrementally all minimal transversals of $\mathcal{H}$.

```
r ← r(H); G ← ∅;
loop /* G ⊆ Tr(H) */
  if (there exists a T ∈ Tr(G)|_r − H) then
    minimize T̄ to a minimal transversal T_1 of H;
  else /* H ⊆ Tr(G), G ⊆ Tr(H) */
    if (there exists G' ⊆ G with |G'| = r + 1 such that
          for all E ∈ G, E ⊈ {x ∈ V : d(x, G') > 1})
    then
      minimize {x ∈ V : d(x, G') > 1} to a minimal transversal T_1 of H;
    else /* G = Tr(H) */
      exit loop;
    fi;
  fi;
  output(T_1); G ← G ∪ {T_1};
endloop;
```

FIG. 1. *Algorithm computing $Tr(\mathcal{H})$ in incremental-polynomial time if $r(\mathcal{H}) \leq k$, for constant $k \geq 0$.*

CLAIM F. $\overline{T} \notin Cov(\mathcal{G})$.

Proof. Assume to the contrary that $\overline{T} \in Cov(\mathcal{G})$. Assume, furthermore, that there exists $G \in \mathcal{G}$ such that $G \subseteq \overline{T}$. This implies $G \cap T = \emptyset$, which means that T is not a transversal of $\mathcal{G}$. However, by definition, $T \in Tr(\mathcal{G})|_r - \mathcal{H}$, and thus T is a transversal of $\mathcal{G}$. We have reached a contradiction, and thus G does not exist. Assume otherwise that there exists $G \in \mathcal{G}$ such that $\overline{T} \subset G$. This implies that $G \notin Tr(\mathcal{H})$, which contradicts $\mathcal{G} \subseteq Tr(\mathcal{H})$. This shows that $\overline{T} \notin Cov(\mathcal{G})$.

By Claims E and F, $\overline{T}$ is a transversal of $\mathcal{H}$ such that $\overline{T} \notin Cov(\mathcal{G})$. Thus $\overline{T}$ contains some $T_1 \in Tr(\mathcal{H})$ such that $T_1 \notin \mathcal{G}$; in fact, every $T_1 \in Tr(\mathcal{H})$ such that $T_1 \subseteq \overline{T}$ has this property.

(ii) Assume that **Q** is false. Hence, there exists $\mathcal{G}' \subseteq \mathcal{G}$ with $|\mathcal{G}'| = r + 1$ such that for every $E \in \mathcal{G}$ we have $E \not\subseteq \{x \in V : d(x, \mathcal{G}') > 1\}$. Note that this is only possible if $|\mathcal{G}| > r(\mathcal{H})$, which implies that $r(\mathcal{H}) \geq 2$. By Lemma 5.1, it follows that $r(Tr(\mathcal{G})) > r$, thus $\mathcal{H} \neq Tr(\mathcal{G})$.

CLAIM G. $T = \{x \in V : d(x, \mathcal{G}') > 1\}$ *is a transversal of* $\mathcal{H}$ *and* $T \notin Cov(\mathcal{G})$.

Proof. Note that if T is a transversal of $\mathcal{H}$, then $T \notin Cov(\mathcal{G})$, as no edge of $\mathcal{G}$ is contained in T and, on the other hand, for no $E \in \mathcal{G}$ does it hold that $T \subset E$ as $E \in Tr(\mathcal{H})$. Thus to prove our claim, it remains to show that T is a transversal of $\mathcal{H}$. Assume that T is not a transversal of $\mathcal{H}$. Hence there exists some $E \in \mathcal{H}$ such that $E \cap T = \emptyset$, i.e., $E \subseteq \overline{T}$. However, this is impossible: since T contains no edge of $\mathcal{G}$, $\overline{T}$ is a transversal of $\mathcal{G}$, and thus also a transversal of $\mathcal{G}'$. From the definition of T, each vertex of $\overline{T}$ appears in at most one edge of $\mathcal{G}'$. Since $\mathcal{G}'$ has $r + 1$ edges, it follows that no transversal T' of $\mathcal{G}'$ with $T' \subseteq \overline{T}$ can have fewer than $r + 1$ vertices. Check that every $H \in \mathcal{H}$ is a transversal of $\mathcal{G}'$; since $|H| \leq r$, it follows that $E \not\subseteq \overline{T}$, which is a contradiction. Thus T is a transversal of $\mathcal{H}$, and the claim is proved.

Since $T \notin Cov(\mathcal{G})$, T contains some $T_1 \in Tr(\mathcal{H})$ such that $T_1 \notin Cov(\mathcal{G})$; in fact, every $T_1 \in Tr(\mathcal{H})$ such that $T_1 \subseteq T$ has this property.

Utilizing this result, the algorithm **Trans** in Fig. 1 incrementally outputs the minimal transversals of a simple, nonempty hypergraph $\mathcal{H}$. The correctness of **Trans** can be easily proved from (1) and the described additional transversal determination; note that $\mathcal{G} \subseteq Tr(\mathcal{H})$ is a loop invariant.

To prove the theorem, assume that $r(\mathcal{H}) \leq k$. If $\mathcal{H}$ is not simple, compute $\min(\mathcal{H})$ in polynomial time and proceed with that hypergraph. Since k is a constant, the conditions of

both **if** statements can be checked in time polynomial in the size of $\mathcal{G}$, $\mathcal{H}$, and V. n^k is an upper bound on the number of vertex sets to check in the outer **if**, and $|\mathcal{G}|^{k+1}$ is an upper bound for the number of subhypergraphs $\mathcal{G}' \subseteq \mathcal{G}$ (each of them computable in polynomial time) to test in the inner **if**. In both cases, all tests are clearly polynomial. Furthermore, T_1 is computable in polynomial time (cf. §2). Thus an execution of the loop body needs time polynomial in the size of $\mathcal{G}$, $\mathcal{H}$, and n. Every pass of the loop but the last yields an additional minimal transversal T_1 of $\mathcal{H}$ for output. The result follows. □

The following result is an immediate consequence of the last theorem.

COROLLARY 5.3. TRANS-HYP *is polynomial if* $r(\mathcal{H}) \leq k$ *or* $r(\mathcal{G}) \leq k$ *for fixed* $k \geq 0$.

Note that Theorem 5.2 generalizes the well-known result that the maximal independent sets of a graph are output-efficiently computable to hypergraphs with edge-size bounded by a constant. (Recall that a maximal independent set is the complement of a minimal transversal.) Thus the question in [29] of whether there is an output-polynomial total time algorithm for generating all maximal independent sets of a hypergraph is affirmatively answered for hypergraphs of bounded edge-size.

Let us now consider a constant upper bound on the edge size. In this case the computation of the transversal hypergraph is efficiently possible even in the input size.

THEOREM 5.4. *If* $ar(\mathcal{H}) \geq n - k$ *for fixed* $k \geq 0$, *then* $Tr(\mathcal{H})$ *is computable in input-polynomial time.*

Proof. Without loss of generality we assume that $\mathcal{H}$ is simple. We know from Proposition 4.7(iii) that $Tr(\mathcal{H}) \leq \overline{\delta(\mathcal{H})}$. Since $r(\overline{\delta(\mathcal{H})}) \leq k+1$, only the sets $V' \subseteq V$ of up to $k+1$ elements are candidates for minimal transversals, and there are no more than roughly n^{k+1} of them. □

From Theorems 5.4 and 4.8, we immediately have the following result.

COROLLARY 5.5. TRANS-HYP *and* SIMPLE-H-SAT *are polynomial if* $ar(\mathcal{H}) \geq n - k$ *for fixed* $k \geq 0$.

As the third restriction, let us consider hypergraphs where the edges differ in their size by at most some given constant. We note a simple but important lemma.

LEMMA 5.6. *Let* $\mathcal{H}$ *be a hypergraph on* V *and* $a = ar(\mathcal{H})$, $r = r(\mathcal{H})$. *Then* $\mathcal{H}$ *is not saturated if and only if there exists a set* $V' \subseteq V$, $a \leq |V'| \leq r$, *such that* $V' \notin Cov(\mathcal{H})$.

Proof. Assume there exists $V' \subseteq V$ such that $V' \notin Cov(\mathcal{H})$. If $|V'| < a$, we may add any vertices $v_1, \ldots, v_{a-|V'|}$ from $V - V'$ to V', and $V' \cup \{v_1, \ldots, v_{a-|V'|}\} \notin Cov(\mathcal{H})$ will hold. If $|V'| > r$, we may remove any vertices $v_1, \ldots, v_{|V'|-r}$ from V' without establishing $V' - \{v_1, \ldots, v_{|V'|-r}\} \in Cov(\mathcal{H})$. Thus the *only if* direction holds. The *if* direction is trivial. □

THEOREM 5.7. H-SAT *is polynomial if* $r(\mathcal{H}) - ar(\mathcal{H}) \leq k$ *for fixed* $k \geq 0$.

Proof. Let $a = ar(\mathcal{H})$ and $r = r(\mathcal{H})$. There is a simple saturation algorithm which is polynomial for this restriction: check for each $X \subseteq V$, $|X| = a, a+1, \ldots, r$, whether $X \notin Cov(\mathcal{H})$; output "no" if the first such X is found and "yes" if none are found. By Lemma 5.6, this algorithm is correct.

Let us analyze the complexity of this algorithm. Consider $Y \in \mathcal{H}$. The exact number of the sets X covered by Y is given by

$$C(Y, a, r) = \sum_{i=0}^{r-|Y|} \binom{n-|Y|}{i} + \sum_{i=0}^{|Y|-a} \binom{|Y|}{i} - 1, \tag{2}$$

where the first (resp., second) term on the right-hand side of the equation equals the number of the sets X covered as supersets (resp., subsets). The number $C(Y, a, r)$ is bounded by the sum of the number of sets $X \subseteq V$ with $|X| \leq r$ that a fixed $Y' \subseteq V$ with $|Y'| = a$ covers as superset and the number of sets $X \subseteq V$ with $|X| \geq a$ that a fixed $Y' \subseteq V$ with $|Y'| = r$

covers as subset. Let $c = r - a + 1$. Thus

$$\begin{aligned} C(Y, a, r) &\le \sum_{i=0}^{r-a} \left[\binom{n-a}{i} + \binom{r}{i} \right] \\ &\le 2c \cdot \max \left\{ \binom{n-a}{i}, \binom{r}{i} : 0 \le i \le c-1 \right\} \\ &\le 2cn^{c-1}. \end{aligned}$$

The last inequality uses the facts that $u \le v$ implies $\binom{u}{w} \le \binom{v}{w}$ and $\binom{u}{w} \le u^w$. As $c \le k+1$, we have $C(Y, a, r) \le s(n)$ for some polynomial $s(n)$. Hence $\mathcal{H}$ covers at most $m \cdot s(n)$ of the sets X.

Note that the subsets of V of size i can be systematically output with polynomial-time delay $p(n)$ (cf. [12] for a suitable order). Moreover, testing $X \in Cov(\mathcal{H})$ can be done in polynomial time $q(m, n)$.

Hence, if $\mathcal{H}$ is not saturated, the algorithm finds the first X such that $X \notin Cov(\mathcal{H})$ in time $O(m \cdot s(n)(p(n) + q(m, n)))$, i.e., in polynomial time. If $\mathcal{H}$ is saturated, then all sets X are checked. Let $f(n, a, r) = \binom{n}{a} + \binom{n}{a+1} + \cdots + \binom{n}{r}$ denote their number. Since $\mathcal{H}$ is saturated, it covers each of the sets X. On the other hand, $\mathcal{H}$ covers at most $m \cdot s(n)$ of the sets X; consequently, $f(n, a, r) \le m \cdot s(n)$. This ensures that the algorithm again terminates in time $O(m \cdot s(n)(p(n) + q(m, n)))$. □

COROLLARY 5.8 (to Theorem 5.7). SIMPLE-H-SAT *is polynomial if* $r(\mathcal{H}) - ar(\mathcal{H}) \le k$ *for fixed* $k \ge 0$.

5.2. Acyclic hypergraphs. As in graph theory, the notion of acyclicity is appealing in hypergraph theory from a theoretical as well as a practical point of view. In some contexts, acyclic hypergraphs gain special attention. For example, in relational database theory acyclic hypergraphs (also called *tree schemes*) were introduced in [8] and later used by many authors (cf. [53], [30]), especially in relational database design; cf. [2], [4], [5], [17].

Several **NP**-complete problems on hypergraphs become polynomial for acyclic hypergraphs [57]. Since it is not straightforward to carry over the definition of a cycle from graphs to hypergraphs, there are many notions of acyclicity in a hypergraph; cf. [16], [18], [7]. We refer to α-, β-, γ-, and *Berge*-acyclicity as stated in [16], where the proper inclusion hierarchy *Berge*-acyclic $\Rightarrow$ γ-acyclic $\Rightarrow$ β-acyclic $\Rightarrow$ α-acyclic is proved.

The notion of α-acyclicity came up in the context of relational database theory [23], [58], [5], [16]. A hypergraph $\mathcal{H}$ is *α-acyclic* iff $\mathcal{H} = \emptyset$ or $\mathcal{H}$ is by the Graham–Yu–Ozsoyoglu (GYO) reduction (from Graham [23] and (independently) Yu and Ozsoyoglu [58]), that is, by repeated application of one of the following two rules:

(1) if vertex v occurs in only one edge E, remove v from E;

(2) if distinct edges E, E' satisfy $E' \subseteq E$, remove E',

reducible to the hypergraph $\{\emptyset\}$.

For example, the hypergraph in Fig. 2 is α-acyclic. Note that α-acyclicity of a hypergraph $\mathcal{H}$ can be checked with an algorithm of Tarjan and Yannakakis [50] in time $O(n + t)$, where n is the number of vertices and t is the total size of the edges of $\mathcal{H}$.

Note that rule (2) implies that a hypergraph $\mathcal{H}$ is α-acyclic if and only if $\max(\mathcal{H})$ is α-acyclic. Consequently, an α-acyclic hypergraph $\mathcal{H}$ may contain an α-cyclic subhypergraph $\mathcal{H}' \subseteq \mathcal{H}$. For example, in Fig. 2 let be $\mathcal{H}' = \{\{a, b, c\}, \{c, d, e\}, \{a, e, f\}\}$. However, this is not what one expects from acyclicity, as it is surprising that a cycle disappears by adding edges. Fagin points out this anomaly [16] and introduces the more natural concept of β-acyclicity.

DEFINITION 5.2. *A hypergraph $\mathcal{H}$ is β-acyclic iff every subhypergraph $\mathcal{H}' \subseteq \mathcal{H}$ is α-acyclic.*

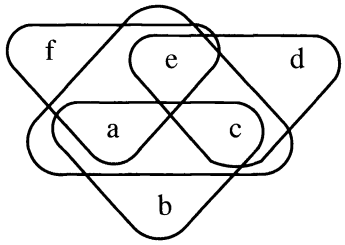


FIG. 2. *α-acyclic but β-cyclic hypergraph $\mathcal{H} = \{\{a, b, c\}, \{c, d, e\}, \{a, e, f\}, \{a, c, e\}\}$.*

The hypergraph in Fig. 2 is β-cyclic, i.e., not β-acyclic. In [16] Fagin presents various equivalent definitions of β-acyclicity, among them a related acyclicity criterion by Graham [23], and he also gives a polynomial-time algorithm to test β-acyclicity.

Our next aim is to show that the transversal hypergraph of a β-acyclic hypergraph is efficiently computable with respect to the input and output size.

DEFINITION 5.3. *Let $\mathcal{H}$ be a hypergraph on V. For every $V' \subseteq V$, the partial hypergraph of $\mathcal{H}$ generated by V' is $\mathcal{H}_{V'} = \{E \cap V' | E \in \mathcal{H}\}$.*

For convenience, let us call any vertex an *ear node* of the hypergraph $\mathcal{H}$ if it occurs in exactly one edge of $\mathcal{H}$ (cf. [53, p. 698]). We observe the following rule for transversal computation.

LEMMA 5.9. *Let $\mathcal{H}$ be a simple hypergraph on V with an ear node v that occurs in edge E. Then $\{\{v\} \cup T : T \in Tr(\mathcal{H}_{\overline{E}} - \{\emptyset\})\} \cup Tr(\mathcal{H}_{V-\{v\}})$ is a partitioning of $Tr(\mathcal{H})$.*

Proof. If $\mathcal{H}$ is simple, then every essential vertex $v \in V$ occurs in at least one minimal transversal of $\mathcal{H}$. (This is readily shown from Proposition 2.2.) However, $Tr(\mathcal{H}_{\overline{E}} - \{\emptyset\})$ is never empty. Now consider $Tr(\mathcal{H})$. The minimal transversals can be partitioned into $Tr(\mathcal{H}) = \mathcal{T}_1 \cup \mathcal{T}_2$, where $\mathcal{T}_1$ contains all minimal transversals that contain v and $\mathcal{T}_2$ contains all others. Since $Tr(Tr(\mathcal{H})) = \mathcal{H}$ if $\mathcal{H}$ is simple (Proposition 2.2), E is a minimal transversal of $Tr(\mathcal{H})$. It follows that $E \cap T = \{v\}$ for every $T \in Tr(\mathcal{H})$ that contains v. Indeed, if $\{v\} \subset E \cap T$ would hold then, since E is the only edge of $\mathcal{H}$ that contains v, $T - \{v\}$ would be a transversal of $\mathcal{H}$, which contradicts $T \in Tr(\mathcal{H})$. Thus $E \cap T = \{v\}$. This implies $T - \{v\} \subseteq \overline{E}$. From this it is clear that $\mathcal{T}_1 = \{\{v\} \cup T : T \in Tr(\mathcal{H}_{\overline{E}} - \{\emptyset\})\}$. It is immediate that $\mathcal{T}_2 = Tr(\mathcal{H}_{V-\{v\}})$. □

THEOREM 5.10. *The minimal transversals of a β-acyclic hypergraph $\mathcal{H}$ are P-enumerable.*

Proof. Recall that a problem is P-enumerable if there is some algorithm which computes all solutions to the problem in time $p(IS)N$, where p is some polynomial in the input size IS and N is the number of solutions. To include the case in which the problem has *no* solution, we slightly modify this convention from $p(IS)N$ to $p(IS)(N + 1)$; this modification is not substantial.

To prove the theorem, we proceed as follows: first we note some facts regarding β-acyclic hypergraphs and then we give an algorithm for transversal computation. We may assume without loss of generality that $\mathcal{H}$ has no inessential vertices and $\mathcal{H}$ is simple, since $\min(\mathcal{H})$ is polynomial-time computable, $Tr(\mathcal{H}) = Tr(\min(\mathcal{H}))$, and $\min(\mathcal{H})$ is β-acyclic if $\mathcal{H}$ is β-acyclic.

There is a simple observation on ear nodes of β-acyclic hypergraphs.

FACT 1. *Every simple β-acyclic hypergraph $\mathcal{H}$ that contains a nonempty edge has an ear node.*

Indeed, if $\mathcal{H}$ is β-acyclic, then it is also α-acyclic, and the GYO reduction must succeed. However, as $\mathcal{H}$ is simple, rule (2) is not applicable, and as $\mathcal{H} \neq \{\emptyset\}$, rule (1) must apply. Hence $\mathcal{H}$ has an ear node.

Algorithm BetaTr($\mathcal{H}$)
input: Simple β-acyclic hypergraph $\mathcal{H}$.
output: $Tr(\mathcal{H})$.

if $\mathcal{H} = \emptyset$ **then return**($\{\emptyset\}$)
else
  if $\mathcal{H} = \{\emptyset\}$ **then return**($\emptyset$)
  else
    $v \leftarrow$ an ear node in $E \in \mathcal{H}$;
    $\mathcal{H}_1 \leftarrow \min(\mathcal{H}_{\overline{E}} - \{\emptyset\})$; $\mathcal{H}_2 \leftarrow \min(\mathcal{H}_{V-\{v\}})$;
    $\mathcal{T}_1 \leftarrow$ **BetaTr**($\mathcal{H}_1$); $\mathcal{T}_2 \leftarrow$ **BetaTr**($\mathcal{H}_2$);
    return($\{T \cup \{v\} : T \in \mathcal{T}_1\} \cup \mathcal{T}_2$)
  fi;
fi;

FIG. 3. *Output-polynomial algorithm for computing $Tr(\mathcal{H})$ of a simple, β-acyclic hypergraph $\mathcal{H}$.*

FACT 2. *Let $\mathcal{H}$ be a β-acyclic hypergraph and $V' \subseteq V$. Then $\mathcal{H}_{V'}$ is β-acyclic.* (See [16, p. 530].)

Now consider the procedure **BetaTr** in Fig. 3, where the input hypergraph $\mathcal{H}$ is simple. For every hypergraph, let $\mathcal{G}$ denote $v_e(\mathcal{G}) = |V_e(\mathcal{G})|$ in what follows.

The correctness of **BetaTr** is shown by induction on $v_e(\mathcal{H})$. If $v_e(\mathcal{H}) = 0$, then $\mathcal{H} = \emptyset$ or $\mathcal{H} = \{\emptyset\}$, and the result is correct. Now consider $v_e(\mathcal{H}) > 0$ and assume the hypothesis is correct for simple hypergraphs with fewer essential nodes than $\mathcal{H}$. As $\mathcal{H}$ is simple, β-acyclic, and $v_e(\mathcal{H}) > 0$, by Fact 1, $\mathcal{H}$ has an ear node; hence a vertex v in edge E will be found. From Lemma 5.9, we have

$$Tr(\mathcal{H}) = \{\{v\} \cup T : T \in Tr(\mathcal{H}_{\overline{E}} - \{\emptyset\})\} \cup Tr(\mathcal{H}_{V-\{v\}}),$$

which is equivalent to

$$Tr(\mathcal{H}) = \{\{v\} \cup T : T \in Tr(\min(\mathcal{H}_{\overline{E}} - \{\emptyset\}))\} \cup Tr(\min(\mathcal{H}_{V-\{v\}})).$$

Since a hypergraph is β-acyclic iff every subhypergraph of it is β-acyclic, in conjunction with Fact 2, it follows that $\mathcal{H}_1$ and $\mathcal{H}_2$ are simple, β-acyclic hypergraphs. Because $\mathcal{H}_1$ and $\mathcal{H}_2$ have fewer essential vertices than $\mathcal{H}$, by the induction hypothesis we have $Tr(\mathcal{H}_i) =$ **BetaTr**$(\mathcal{H}_i) = \mathcal{T}_i$ for $i = 1, 2$. Thus

$$Tr(\mathcal{H}) = \{\{v\} \cup T : T \in Tr(\mathcal{H}_{\overline{E}} - \{\emptyset\})\} \cup Tr(\mathcal{H}_{V-\{v\}}) = \{\{v\} \cup T : T \in \mathcal{T}_1\} \cup \mathcal{T}_2.$$

This is exactly what **BetaTr**($\mathcal{H}$) returns if $V_e(\mathcal{H}) \neq \emptyset$; hence the claimed statement holds for $\mathcal{H}$.

It remains to prove the claim on the complexity of **BetaTr**. Denote by *calls*($\mathcal{H}$) the total number of recursive calls to **BetaTr** in computing **BetaTr**($\mathcal{H}$). We show by induction on $v_e(\mathcal{H})$ that

$$calls(\mathcal{H}) \leq 2v_e(\mathcal{H}) \cdot |Tr(\mathcal{H})|.$$

For $v_e(\mathcal{H}) = 0$, we have either $\mathcal{H} = \emptyset$ or $\mathcal{H} = \{\emptyset\}$; the statement clearly holds. Now consider the case $v_e(\mathcal{H}) > 0$ and assume the statement is correct for hypergraphs with fewer essential vertices. We have that

$$calls(\mathcal{H}) = calls(\mathcal{H}_1) + calls(\mathcal{H}_2) + 2.$$

By the induction hypothesis, we obtain that

$$calls(\mathcal{H}_1) \leq 2v_e(\mathcal{H}_1) \cdot |Tr(\mathcal{H}_1)| \leq 2(v_e(\mathcal{H}) - 1) \cdot |Tr(\mathcal{H}_1)| \leq 2v_e(\mathcal{H}) \cdot |Tr(\mathcal{H}_1)| - 2;$$

the last inequation holds because $\mathcal{H}_1 \neq \{\emptyset\}$, and therefore $|Tr(\mathcal{H}_1)| > 0$. Similarly, we obtain by the induction hypothesis that

$$calls(\mathcal{H}_2) \leq 2v_e(\mathcal{H}_2)\cdot|Tr(\mathcal{H}_2)| \leq 2v_e(\mathcal{H})\cdot|Tr(\mathcal{H}_2)|.$$

Thus

$$\begin{aligned} calls(\mathcal{H}) &= calls(\mathcal{H}_1) + calls(\mathcal{H}_2) + 2 \\ &\leq 2v_e(\mathcal{H})\cdot|Tr(\mathcal{H}_1)| - 2 + 2v_e(\mathcal{H})\cdot|Tr(\mathcal{H}_2)| + 2 \\ &= 2v_e(\mathcal{H})\cdot|Tr(\mathcal{H}_1)| + 2v_e(\mathcal{H})\cdot|Tr(\mathcal{H}_2)|. \end{aligned}$$

Now, since $|Tr(\mathcal{H})| = |Tr(\mathcal{H}_1)| + |Tr(\mathcal{H}_2)|$, it follows that $calls(\mathcal{H}) \leq 2v_e(\mathcal{H})\cdot|Tr(\mathcal{H})|$, i.e., the claimed statement holds.

The body of **BetaTr** can be clearly executed in polynomial time. Thus, the total run time of **BetaTr**$(\mathcal{H})$ is bounded by $p(m,n)\cdot(calls(\mathcal{H}) + 1)$ for some polynomial p in m and n (i.e., the number of edges and vertices of $\mathcal{H}$). Since

$$p(m,n)\cdot(calls(\mathcal{H}) + 1) \leq p(m,n)\cdot(2v_e(\mathcal{H})\cdot|Tr(\mathcal{H})| + 1),$$

it follows that the run time of **BetaTr**$(\mathcal{H})$ is bounded by $q(m,n)\cdot(|Tr(\mathcal{H})| + 1)$ for some polynomial q. Thus **BetaTr**$(\mathcal{H})$ P-enumerates the minimal transversals of $\mathcal{H}$. The result follows. □

We remark that it is not difficult to modify **BetaTr** such that the minimal transversals are output with polynomial-time delay; we leave this to the reader.

From Theorem 5.10, we immediately obtain, together with Theorems 4.5 and 4.12, the following polynomial cases of the transversal problem and the saturation problem.

COROLLARY 5.11. TRANS-HYP *and* SIMPLE-H-SAT *are polynomial for* β*-acyclic hypergraphs.*

6. Overview of related problems and applications. In this section we give a short overview of some related problems and applications in different fields of computer science. For space reasons the exposition is rather succinct. In particular, we omit several formal definitions and all proofs. A full discussion, exact definitions, all proofs, and more material can be found in the extended report [15]; other interesting problems related to TRANS-HYP have recently been studied in [32].

6.1. Clause satisfiability. We have identified the following restrictions of the well-known SATISFIABILITY problem which are $\leq_m^p$-equivalent to TRANS-HYP (resp., co-TRANS-HYP).

Problem. INTERSECTING MONOTONE SAT (IMSAT).

Instance. A set $\mathcal{C}$ of clauses such that each clause is either positive (i.e., consists entirely of positive literals) or negative (i.e., consists entirely of negative literals) and for each positive clause C_1 and negative clause C_2 of $\mathcal{C}$, there exists an atom u such that $u \in C_1$ and $\neg u \in C_2$.

Question. Is $\mathcal{C}$ satisfiable?

Problem. SYMMETRIC INTERSECTING MONOTONE SAT (SIMSAT)

Instance. Restriction of IMSAT to instances $\mathcal{C}$, where the negative clauses are precisely all clauses C^- such that $C^- = \{\neg u : u \in C^+\}$ for some positive clause $C^+ \in \mathcal{C}$. (By this restriction, nonempty positive clauses of $\mathcal{C}$ are mutually intersecting.)

Question. Is $\mathcal{C}$ satisfiable?

Both problems are $\leq_m^p$-equivalent to co-SIMPLE-H-SAT. Interestingly, unlike SATISFIABILITY or most known other restrictions of SATISFIABILITY, these problems become polynomial-time decidable as soon as the cardinality of the clauses in the problem instances is bounded by a constant.

6.2. Design of relational databases. For the basic concepts of relational database theory consult [39] or [53]. If F is a set of functional dependencies (FDs) on a set of attributes U, then F^+ denotes the closure of F, i.e., all those dependencies on U that follow from F. Let $RS = \langle U, F \rangle$ be a relation schema, where U is a finite set of attributes and F is a set of functional dependencies. A relation instance R over U is an *Armstrong relation* for RS iff the functional dependencies that hold in R are precisely F^+. It was recently advocated that Armstrong relations can be used as a very profitable tool in database design [44], [40]. In this context it is important to compute an Armstrong relation from a given set of functional dependencies and vice versa. The following related decision problem is $\leq_m^p$-equivalent to SIMPLE-H-SAT.

Problem. FD-RELATION EQUIVALENCE.

Instance. A relation instance R and a set F of FDs in *Boyce–Codd normal form* (BCNF), both on a set of attributes U.

Question. Is R an Armstrong relation for F?

A relation scheme $RS = \langle U, F \rangle$ is in BCNF if, for every $X \to Y \in F^+$ such that $Y \not\subseteq X$, X is a superkey for RS, i.e., $X \to U$.

Using Corollary 5.3, it can be shown that the problem FD-RELATION EQUIVALENCE becomes polynomial if F is in BCNF and, for each $X \to Y$ of F, X contains fewer than k attributes for some constant k. Moreover, we can show that, under the same restrictions, the problem of *generating* an Armstrong relation for F can be done in output-polynomial time.

A relation instance R on U is in BCNF if the schema $\langle U, F_R \rangle$ is in BCNF, where F_R is the set of all FDs that hold on R. The following problem can be shown to be $\leq_m^p$-equivalent to co-SIMPLE-H-SAT.

Problem. ADDITIONAL KEY for relation instances.

Instance. A relation instance R on attributes U, a set $\mathcal{K}$ of minimal keys for R.

Question. Is there a minimal key for the scheme $RS = \langle U, F_R \rangle$ not contained in $\mathcal{K}$?

Note that the additional key problem for relation *schemes* is polynomial-time solvable by an algorithm of Lucchesi and Osborn [37].

6.3. Updates in distributed databases. In distributed databases, mutual exclusion of groups of sites, which is necessary for executing critical operations, can be realized by defining a priori a set of groups (quorums) that intersect each other [33]. A group of sites can perform the critical operation only if it contains a quorum of this set. Clearly, it is natural to consider only minimal quorums, i.e., no quorum should properly contain any other quorum. In terms of hypergraph theory, this means that the specified quorums should constitute a simple, intersecting hypergraph on the set of sites, termed a *coterie* in [20]. For example, the hypergraph $\mathcal{C} = \{\{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{b, c, d\}\}$ is a coterie on vertices $\{a, b, c, d\}$. Coteries appear to be a fruitful concept for access control in distributed databases; cf. [20], [25], [19].

A coterie $\mathcal{A}$ *dominates* a coterie $\mathcal{B}$ if and only if $\mathcal{A} \neq \mathcal{B}$ and, for each $B \in \mathcal{B}$, there exists an $A \in \mathcal{A}$ such that $B \supseteq A$. A coterie $\mathcal{A}$ is *nondominated* (ND) if there is no coterie $\mathcal{B}$ that dominates $\mathcal{B}$. In practice, ND coteries are desired for decision agreement groups with respect to reliability considerations. Unfortunately, no efficient test for nondomination of a coterie is known; in [20] an algorithm which has worst-case run time exponential in the

number of vertices is described. Interestingly, this problem turns out to be an instance of the SELF-TRANSVERSALITY problem.

THEOREM 6.1. *A coterie C is ND if and only if $C = Tr(C)$.*

Since every self-transversal hypergraph is a coterie, we get from Theorems 6.1 and 4.13 the following corollary.

COROLLARY 6.2. *Checking whether a coterie is ND* (ND COTERIE) *is $\leq_m^p$-equivalent to* SIMPLE-H-SAT.

Note that by Theorem 6.1 and Corollary 5.3 ND COTERIE is polynomial for small quorums.

In the extended report, we also show that for a recent generalization of this approach to bicoteries and semicoteries, which model read and write operations by read quorums and write quorums [19], [25], the complexity of testing nondomination is not increased and remains $\leq_m^p$-equivalent to TRANS-HYP.

6.4. Boolean switching theory. We assume that the reader is familiar with the basic concepts of switching circuits and Boolean functions (BFs); for details we refer to the standard literature (e.g., [56]). The design of efficient algorithms for the computation of prime implicants from a function table or a logical expression has been a topic of research over decades.

It is well known that every monotone Boolean function has a unique *minimal conjunctive normal form* consisting of a conjunction of disjunctions of atoms where no conjunct subsumes any other.

In [15] we show the following results. Given a monotone Boolean expression E and a set P of prime implicants of E, it is **NP**-complete to determine whether there exists an additional prime implicant of E. However, if E is in minimal conjunctive normal form, then the same problem is $\leq_m^p$-equivalent to co-SIMPLE-H-SAT. In particular, in the latter case, the prime implicants of E are precisely the minimal transversals of the conjuncts of E. Thus, the polynomial subcases for computing all minimal hypergraph transversals described in §5 carry over to the problem of computing the prime implicants of a monotone Boolean function in conjunctive normal form. This problem is also investigated in [9], which gives some new results.

6.5. Model-based diagnosis. Basic techniques for model-based diagnosis have been developed within AI by de Kleer and Williams [13] and Reiter [45]. To introduce the necessary concepts briefly, a *system* is a pair $(SD, COMP)$, where SD, the *system description*, is a set of usually first-order sentences and $COMP$ is a set of constants which model the system *components*. This general system description is used together with a set OBS of first-order sentences, which are particular *observations* on the system behavior, to diagnose faults. The system description SD makes use of a distinguished predicate $AB(c)$ which interprets "component c operates in abnormal mode." Now a *diagnosis* for $(SD, COMP, OBS)$ is a minimal set $\Delta \subseteq COMP$ of components such that

$$\mathcal{T} = SD \cup OBS \cup \{AB(c) \mid c \in \Delta\} \cup \{\neg AB(c) \mid c \in COMP - \Delta\}$$

is consistent.[4] Of course, if there is no fault, $\Delta = \emptyset$ must be a diagnosis, otherwise SD is not a sound system description.

In [45] a characterization of diagnoses in terms of so called conflict sets is given. A *conflict set* is a set $C \subseteq COMP$ such that $SD \cup OBS \cup \{\neg AB(c) \mid c \in C\}$ is inconsistent. A conflict set C is minimal if and only if no proper subset of C is a conflict set. The fundamental theorem on conflict sets and diagnoses in [45] put into hypergraph terminology is as follows.

[4] It is presupposed that $\mathcal{T}$ is always in a decidable subclass of first-order predicate calculus.

THEOREM 6.3 ([45]). $\Delta \subseteq COMP$ *is a* diagnosis *for* $(SD, COMP, OBS)$ *if and only if* $\Delta \in Tr(\mathcal{C})$, *where* $\mathcal{C}$ *denotes the set of all minimal conflict sets of* $(SD, COMP, OBS)$.

From Theorems 6.3 and 4.12 we immediately have the following theorem.

THEOREM 6.4. *Let* $\mathcal{C}$ *be the set of minimal conflict sets of* $(SD, COMP, OBS)$ *and* $\mathcal{D}$ *be a set of diagnoses for* $(SD, COMP, OBS)$. *Given* $\mathcal{C}$ *and* $\mathcal{D}$ *for input, deciding if there is an additional diagnosis not contained in* $\mathcal{D}$ *is* $\leq_m^p$*-equivalent to co-*SIMPLE-H-SAT.

The determination of diagnoses from the given minimal conflict sets is essential in popular algorithms for model-based diagnosis [45], [13]. Deciding if an already computed set of diagnoses is complete with respect to a given set of minimal conflict sets (i.e., it consists of all diagnoses) is an important subproblem if diagnoses are computed incrementally. Therefore, the complexity of the additional diagnosis problem is of crucial interest.

7. Conclusion. We have studied two computational problems on hypergraphs in this paper, namely, deciding saturation of a hypergraph (H-SAT) and the recognition of the transversal hypergraph (TRANS-HYP). The latter problem is closely related to computing all minimal transversals of a hypergraph.

The complexity of computing (resp., recognizing) all minimal transversals of a hypergraph is an open problem [20], [41], [14], [29]. We showed that checking saturation for a simple hypergraph (SIMPLE-H-SAT) is computationally equivalent to TRANS-HYP; without the restriction to simple hypergraphs, H-SAT was proved co-**NP**-complete in its general version as well as for various subcases.

In the study of SIMPLE-H-SAT and TRANS-HYP, we investigated the relationships of these problems to well-studied problems such as SATISFIABILITY and HP2C. Several $\leq_m^p$-equivalent subproblems were exhibited, the most important among them being SELF-TRANSVERSALITY. Moreover, narrowing the open problems "frontier," we showed that some generalizations of these two problems are intractable and several important subcases are polynomial. Some of those results use algorithms that compute, under certain restrictions, all minimal transversals of a hypergraph in output-polynomial total time; the most important restriction is probably a constant upper bound on the edge size.

Our results apply to various problems in database theory, switching theory, logic, and AI, which are all closely related to SIMPLE-H-SAT and TRANS-HYP.

For future research, we present the following open questions.

1. What is the complexity of SIMPLE-H-SAT and TRANS-HYP? In particular, are these problems co-**NP**-complete or can they be solved in polynomial time (or, less restrictive, in nondeterministic polynomial time)? In connection with this, are the minimal transversals of a hypergraph $\mathcal{H}$ computable in output-polynomial total time?

2. How do SIMPLE-H-SAT and TRANS-HYP relate to other open problems in **NP**-completeness (cf. [28])? The most famous such problem, GRAPH ISOMORPHISM, has been studied extensively in the literature. It is known that if GRAPH ISOMORPHISM is **NP**-complete and $\mathbf{P} \neq \mathbf{NP}$, then the polynomial hierarchy collapses at level two [48], which is not expected by the experts [27]. However, there seems to be no trivial relation between GRAPH ISOMORPHISM on the one hand and SIMPLE-H-SAT and TRANS-HYP on the other.

3. What do we gain by using randomized algorithms, probabilistic algorithms, or interactive proof systems (cf. [26], [27]) for SIMPLE-H-SAT and TRANS-HYP, and what about weaker forms of reductions than polynomial transformability (especially randomized reductions; cf. [1], [47], [55])? Schöning [47] defines within **NP** a low hierarchy $L_0, L_1, \ldots$, where $L_0 = \mathbf{P}$, $L_1 = \mathbf{NP} \cap$ co-**NP**, and a high hierarchy $H_0, H_1, \ldots$, where $H_0 =$ **NP**-complete, and the other classes $H_i, i > 0$, correspond to weakened versions of **NP**-completeness, such as γ-completeness [1] (fully contained in H_1), etc., which are believed still strong enough that no problem in **P** satisfies any of them. Classifying co-SIMPLE-H-SAT into the low or

high hierarchy would give strong evidence that the problem is not **NP**-complete or intractable, respectively.

Acknowledgments. The authors thank L. Lovász and M. Yannakakis for interesting discussions of this work and valuable comments. They further thank the referees of previous versions of this paper as well as H. Mannila and D. Plaisted for their comments and helpful suggestions.

Note added in proof. (1) In a recent paper, Fredman and Khachiyan showed that $Tr(\mathcal{H})$ can be recognized in time $O(n^{\log n})$ (*On the Complexity of Dualization of Monotone Disjunctive Normal Forms*. Tech. report LCS-TR-225, Department of Computer Science, Rutgers University, 1994). This implies that all problems that are $\leq_m^p$-equivalent to TRANS-HYP can be recognized within the same time bound. Moreover, it strongly suggests that all these problems are not co-**NP**-complete and provides partial evidence to the thesis that TRANS-HYP is close to the border between polynomiality and co-**NP**-hardness.

(2) As pointed out by V. Gurvich, $\leq_m^p$-equivalence between TRANS-HYP and SELF-TRANSVERALITY (Theorem 4.13) was earlier proved by P. D. Seymour in the context of Boolean functions in his master's thesis (see Quart. J. Math. Oxford, 25 (1974), p. 309).

REFERENCES

[1] L. Adleman and K. Manders, *Reducibility, randomness, and intractability*, in Proc. 9th ACM Symposium on the Theory of Computing, 1977, pp. 151-163.

[2] C. Batini, A. D'Atri, and M. Moscarini, *Formal tools for top-down and bottom-up generation of acyclic relational database schemata*, in Proc. 7th International Conference on Graph-Theoretic Concepts in Computer Science, Linz, Austria, 1981.

[3] C. Beeri, M. Down, R. Fagin, and R. Statman, *On the structure of Armstrong relations for functional dependencies*, J. Assoc. Comput. Mach., 31 (1984), pp. 30-46.

[4] C. Beeri, R. Fagin, D. Maier, A. Mendelzon, J. Ullman, and M. Yannakakis, *Properties of acyclic database schemata*, in Proc. 13th ACM Symposium on the Theory of Computing, 1981, pp. 355-362.

[5] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis, *On the desirability of acyclic database schemes*, J. Assoc. Comput. Mach., 30 (1983), pp. 479-513.

[6] C. Benzaken, *Critical hypergraphs for the weak chromatic number*, J. Combin. Theory, Ser. B, 29 (1980), pp. 328-338.

[7] C. Berge, *Hypergraphs*, North Holland Mathematical Library, Vol. 45, Elsevier-North Holland, Amsterdam, 1989.

[8] P. A. Bernstein and D. W. Chiu, *Using semijoins to solve relational queries*, J. Assoc. Comput. Mach., 28 (1981), pp. 25-40.

[9] C. Bioch and T. Ibaraki, *Complexity of dualization and identification of positive boolean functions*, RUTCOR research report RRR 25-93, Rutgers University, 1993; Inform. Comput., to appear.

[10] G. Burosch, J. Demetrovics, and G. O. H. Katona, *The poset of closures as a model of changing databases*, Order, 4 (1987), pp. 127-142.

[11] S. Cook, *A taxonomy of problems with fast parallel algorithms*, Inform. and Control, 64 (1985), pp. 2-22.

[12] D. E. Daykin, J. Godfrey, and A. J. W. Hilton, *Existence theorems for Sperner families*, J. Combin. Theory, Ser. A, 17 (1974), pp. 245-251.

[13] J. de Kleer and B. C. Williams, *Diagnosing multiple faults*, Artif. Intell., 32 (1987), pp. 97-130.

[14] J. Demetrovics and V. D. Thi, *Keys, antikeys and prime attributes*, Ann. Univ. Sci. Budapest. Sect. Comput., 8 (1987), pp. 35-52.

[15] T. Eiter and G. Gottlob, *Identifying the minimal transversals of a hypergraph and related problems*, Tech. report CD-TR 91/16, Christian Doppler Laboratory for Expert Systems, Tech. Univ. Vienna, Austria, January 1991.

[16] R. Fagin, *Degrees of acyclicity for hypergraphs and relational database schemes*, J. Assoc. Comput. Mach., 30 (1983), pp. 514-550.

[17] R. Fagin, A. Mendelzon, and J. Ullman, *A simplified universal relation assumption and its properties*, ACM Trans. Database Systems, 7 (1982), pp. 343-360.

[18] J.-C. Fournier and M. L. Vergnas, *A class of bichromatic hypergraphs*, Ann. Discrete Math., 21 (1984), pp. 21-27.

[19] A. Fu, *Enhancing Concurrency and Availability for Database Systems*, Ph.D. thesis, School of Computer Science, Simon Fraser University, Burnaby, BC, Canada V5A 1S6, 1990.
[20] H. Garcia-Molina and D. Barbara, *How to assign votes in a distributed system*, J. Assoc. Comput. Mach., 32 (1985), pp. 841-860.
[21] M. Garey and D. S. Johnson, *Computers and Intractability—A Guide to the Theory of NP-Completeness*, W. H. Freeman, New York, 1979.
[22] M. R. Garey, D. S. Johnson, and L. Stockmeyer, *Some simplified NP-complete graph problems*, Theoret. Comput. Sci., 1 (1976), pp. 237-267.
[23] M. Graham, *On the universal relation*, Computer Systems Research Group report, University of Toronto, Toronto, Ontario, Canada, 1979.
[24] R. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics*, Addison-Wesley, Reading, MA, 1989.
[25] T. Ibaraki and T. Kameda, *Theory of coteries*, IEEE Trans. Parallel and Distributed Systems, 4 (1993), pp. 779-794.
[26] D. S. Johnson, *The NP-completeness column—an ongoing guide*, J. Algorithms, 5 (1984), pp. 147-160.
[27] ———, *The NP-completeness column—an ongoing guide*, J. Algorithms, 9 (1988), pp. 426-444.
[28] ———, *The NP-completeness column—an ongoing guide*, J. Algorithms, 11 (1989), pp. 144-151.
[29] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou, *On generating all maximal independent sets*, Inform. Process. Lett., 27 (1988), pp. 119-123.
[30] P. Kanellakis, *Elements of Relational Database Theory*, in Handbook of Theoretical Computer Science, Vol. B, J. van Leeuwen, ed., Elsevier-North Holland, Amsterdam, 1990.
[31] R. Karp and A. Wigderson, *A fast parallel algorithm for the maximal independent set problem*, J. Assoc. Comput. Mach., 32 (1986), pp. 762-773.
[32] D. Kavvadias, C. Papadimitriou, and M. Sideri, *On Horn envelopes and hypergraph transversals*, in Proc. 4th International Symposium on Algorithms and Computation (ISAAC-93), W. Ng, ed., Lecture Notes in Computer Science 762, Springer-Verlag, 1993.
[33] L. Lamport, *The implementation of reliable distributed multiprocess systems*, Comp. Networks, 2 (1978), pp. 95-114.
[34] E. Lawler, J. Lenstra, and A. Rinnooy Kan, *Generating all maximal independent sets: NP-hardness and polynomial-time algorithms*, SIAM J. Comput., 9 (1980), pp. 558-565.
[35] E. L. Lawler, *Covering problems: Duality relations and a new method of solution*, SIAM J. Appl. Math., 14 (1966), pp. 1115-1132.
[36] L. Lovász, *Coverings and colorings of hypergraphs*, in Proc. 4th Southeastern Conference on Combinatorics, Graph Theory, and Computing, Utilitas Mathematica Publishing, Winnepeg, MB, Canada, 1973, pp. 3-12.
[37] C. L. Lucchesi and S. Osborn, *Candidate keys for relations*, J. Comput. System Sci., 17 (1978), pp. 270-279.
[38] K. Maghout, *Sur la détermination des nombres de stabilité et du nombre chromatique d'un graphe*, in Comptes Rendus des Séances de l'Académie des Sciences, Vol. 248, Gauthier-Villars, Paris, 1959, pp. 3522-3523.
[39] D. Maier, *The Theory of Relational Databases*, Computer Science Press, Rockville, MD, 1983.
[40] H. Mannila and K.-J. Räihä, *Design by example: An application of Armstrong relations*, J. Comput. System Sci., 22 (1986), pp. 126-141.
[41] ———, *Dependency inference*, in Proc. 13th ACM Conference on Very Large Databases, Brighton, United Kingdom, 1987, pp. 155-158.
[42] ———, *Algorithms for inferring functional dependencies*, Tech. report A-1988-3, Department of Computer Science, University of Tampere, Series of Publ. A, 1988.
[43] ———, *Generating Armstrong databases for sets of functional and inclusion dependencies*, Tech. report, A-1988-7, Department of Computer Science, University of Tampere, 1988.
[44] ———, *Practical algorithms for finding prime attributes and testing normal forms*, in Proc. 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Philadelphia, PA, 1989, pp. 128-133.
[45] R. Reiter, *A theory of diagnosis from first principles*, Artif. Intell., 32 (1987), pp. 57-95.
[46] B. Roy, *Algèbre moderne et Thèorie des graphes*, Vol. II, Dunod, Paris, 1970.
[47] U. Schöning, *A low and a high hierarchy within NP*, J. Comput. System Sci., 27 (1983), pp. 14-28.
[48] ———, *Graph isomorphism is in the low hierarchy*, J. Comput. System Sci., 37 (1988), pp. 312-323.
[49] E. Sperner, *Ein Satz über Untermengen einer endlichen Menge*, Math. Z., XXVII (1928), pp. 544-548.
[50] R. E. Tarjan and M. Yannakakis, *Simple linear time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs*, SIAM J. Comput., 13 (1984), pp. 566-579.
[51] V. D. Thi, *Minimal keys and antikeys*, Acta Cybernet., 7 (1986), pp. 361-371.
[52] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa, *A new algorithm for generating all maximal independent sets*, SIAM J. Comput., 6 (1977), pp. 505-517.
[53] J. D. Ullman, *Principles of Database and Knowledge Base Systems*, Vol. 2, Computer Science Press, Rockville, MD, 1988.
[54] L. G. Valiant, *The complexity of computing the permanent*, Theoret. Comput. Sci., 8 (1979), pp. 189-201.
[55] L. G. Valiant and V. V. Vazirani, *NP is as easy as detecting unique solutions*, Theoret. Comput. Sci., 47 (1986), pp. 85-93.

[56] I. WEGENER, *The Complexity of Boolean Functions*, Wiley-Teubner Series in Computer Science, John Wiley and Teubner, Stuttgart, 1987.

[57] M. YANNAKAKIS, *Algorithms for acyclic database schemes*, in Proc. ACM-IEEE 7th Conference on Very Large Database Systems, Cannes, France, 1982, pp. 82-94.

[58] C. T. YU AND M. OZSOYOGLU, *An algorithm for tree-query membership of a distributed query*, in Proc. IEEE Computer Software and Applications Conference, 1979, pp. 306-312.

SIAM J. COMPUT.
Vol. 24, No. 6, pp. 1305-1312, December 1995

010

ON THE REUSE OF ADDITIONS IN MATRIX MULTIPLICATION*

K. KALORKOTI†

Abstract. We consider the problem of multiplying pairs of matrices by means of quadratic algorithms in terms of the reuse of additions. We show that if such an algorithm is to be significantly faster than the naïve matrix multiplication method then it must reuse additions to a great extent. (For example, any quadratic or bilinear algorithm for $n \times n$ matrix multiplication that does not reuse additions, except when reusing nonscalar steps, requires at least $n^3/8 - n^2/4 + n/8$ arithmetic operations.)

Key words. algebraic complexity, bilinear forms, formula, matrix multiplication

AMS subject classifications. 68Q20, 68Q25, 68Q40

1. Introduction.

Matrix multiplication has played a central role in algebraic complexity theory, starting with Strassen's amazing algorithm [8] for multiplying two $n \times n$ matrices in $O(n^{\lg 7})$ arithmetic operations.

The problem is simply stated: Let k be any field and $X = (x_{ij})$, $Y = (y_{ij})$ be $m \times n$, $n \times p$ matrices of distinct indeterminates over k (i.e., they are "general" matrices). We wish to compute $W = (w_{ij})$, $1 \le i \le m$, $1 \le j \le p$, such that $W = XY$. The fact that the w_{ij} are bilinear forms means that we can compute them using *bilinear algorithms*, i.e., computations of the form

$$
\begin{aligned}
p_1 &= u_1(X) \times v_1(Y),\\
p_2 &= u_2(X) \times v_2(Y),\\
&\vdots\\
p_r &= u_r(X) \times v_r(Y),\\
w_{11} &= l_{11}(p_1, \ldots, p_r),\\
w_{12} &= l_{12}(p_1, \ldots, p_r),\\
&\vdots\\
w_{mp} &= l_{mp}(p_1, \ldots, p_r),
\end{aligned}
\tag{1}
$$

where the u_i, v_i, and l_{ij} are nonzero k-linear combinations of their arguments. If we assume that the field k is infinite, then such algorithms are optimal within a factor of 2 of the nonscalar operations, and within a small constant factor of the number of scalar operations plus the number of indeterminates (Winograd [10], Strassen [9], Borodin and Munro [1]). (An arithmetic operation in an algorithm is said to be *nonscalar* if it is either a multiplication, neither of whose operands is a constant, or a division, whose denominator is not a constant. All other arithmetic steps of an algorithm are said to be *scalar*; see [1] for more background.) Of course in computing the various linear forms we might reuse earlier subresults. Pictorially, the algorithm can be viewed as shown in Fig. 1. The boxes represent arithmetic circuits whose operations are $+$, $-$, or $\times$. Moreover, the circuits are *linear*, i.e., in each multiplication at least one of the arguments is a constant. If each circuit has fanout 1 (i.e., it is a *formula*) then the diagram corresponds directly to (1) and the two methods of presentation have essentially the same size (e.g., measured as the number of arithmetic operations; we give a precise definition below). However, if the circuits are allowed to have arbitrary fanout then the two sizes can

*Received by the editors July 9, 1993; accepted for publication (in revised form) July 8, 1994.

†Department of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ, Scotland, United Kingdom (kk@dcs.ed.ac.uk).

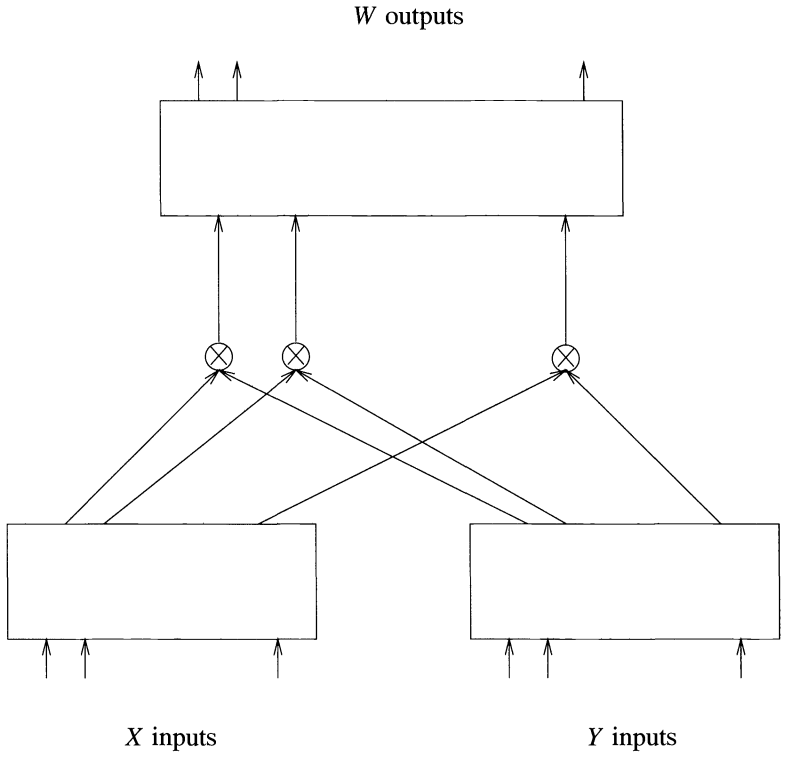


FIG. 1.

be very different. The main purpose of this paper is to show that this difference is quite large for matrix multiplication. Note that any difference in size is a result of *scalar* operations; the number of nonscalar operations (usually called the *rank* of the algorithm) cannot differ.

A very important feature of bilinear algorithms is that they remain valid even if the indeterminates do not commute. In the case of matrix multiplication this fact allows us to use such algorithms recursively by considering the indeterminates to be matrices. If we do not need this feature then we can use *quadratic algorithms*. These have the form (1) but the u_i, v_i are allowed to be k-linear combinations of $X \cup Y$. The gain we make is that quadratic algorithms are optimal with respect to nonscalar operations [9]. Of course the use of quadratic algorithms also enlarges the set of expressions we can compute to include quadratic forms. We will not need the special properties of bilinear algorithms, so we will concentrate on quadratic algorithms; in the diagram we replace X, Y by $X \cup Y$.

The method used in this paper begins by associating a single formula with any given quadratic algorithm. The size of the formula is bounded in terms of the rank of the algorithm and the number of additions (assuming that the algorithm does not reuse scalar operations). This construction is complemented with an algebraic version of Nečiporuk's [5] lower bound on the formula size of Boolean functions. The details are given in §2. Once these results are established, it is an easy matter to derive the main result for matrix multiplication, and this is done in §3.

2. Preliminaries. This section is not specific to matrix multiplication, so we let X be any nonempty finite set of indeterminates over k and assume that we are computing N quadratic

forms $f_1, \ldots, f_N$ by means of the quadratic algorithm $\mathcal{Q}$:

$$
\begin{aligned}
p_1 &= u_1(X) \times v_1(X),\\
p_2 &= u_2(X) \times v_2(X),\\
&\vdots\\
p_r &= u_r(X) \times v_r(X),\\
f_1 &= l_1(p_1, \ldots, p_r),\\
f_2 &= l_2(p_1, \ldots, p_r),\\
&\vdots\\
f_N &= l_N(p_1, \ldots, p_r).
\end{aligned} \tag{2}
$$

By a *formula* F we mean a (connected) arithmetic circuit (with "inputs" from some designated set) of fanout 1 and with a single output, i.e., a tree. The leaves represent the inputs and are labelled by elements of the given input set (which normally includes the constants and indeterminates). The internal vertices (i.e., the nonleaves) are labelled by one of $+$, $-$, or $\times$ and each one computes some result that is a polynomial in the input set with coefficients from k. The output of F is the result of its root. We define the *size* of F as the number of internal vertices and denote it by $||F||$. Since F is a binary tree we have

$$||F|| = \#(\text{leaves of } F) - 1.$$

The *formula size* $||f||$ of a polynomial f is the size of a smallest possible formula with f as its result.

We say that a formula F is *linear* if, in each multiplication, at least one of the arguments is a constant. Thus we can and will view (2) as an algorithm in which each u_i, v_i, l_i is a linear formula with leaves labelled from X and $\{p_1, \ldots, p_r\}$, respectively, as well as constants from k. This corresponds to the situation in which we do not reuse any operations within the three boxes of the diagram. It is clear that a linear formula F with leaves labelled by the elements of a set $G = \{g_1, \ldots, g_s\} \subset k[X] - k$ and constants from k computes an expression of the form $h = \alpha_0 + \alpha_1 g_1 + \cdots + \alpha_s g_s$, where each $\alpha_i \in k$. Now, given the ingredients $\alpha_0, \ldots, \alpha_s$ and $g_1, \ldots, g_s$ for h, we can choose a unique formula H for h by viewing it as $\alpha_0 + (\alpha_1 g_1 + (\alpha_2 g_2 + (\alpha_3 g_3 + \cdots)))$, omitting any zero terms. Note that if the elements of G are linearly independent over k then $\alpha_0, \ldots, \alpha_s$ are determined uniquely by the expression that is computed by F, and thus H is also determined uniquely by F. However, we can always make H depend on F uniquely by fixing a method of expansion for F and we shall assume that this has been done. It is easy to see that the number of additions in H is no more than the number of additions in F (the number of multiplications might be larger). We call H the expanded form of F and set

$$
\begin{aligned}
A_G(F) &= \#(\text{nonzero } \alpha_0, \ldots, \alpha_s \text{ in } H) - 1,\\
M_G(F) &= \#(\alpha_1, \ldots, \alpha_s \text{ in } H \text{ different from } 0, \pm 1).
\end{aligned}
$$

For a sequence $F_1, \ldots, F_t$ we set

$$A_G(F_1, \ldots, F_t) = A_G(F_1) + \cdots + A_G(F_t);$$

we set $M_G(F_1, \ldots, F_t)$ similarly. Note that

$$M_G(F_1, \ldots, F_t) \leq A_G(F_1, \ldots, F_t) + t.$$

We extend these definitions to $\mathcal{Q}$ by

$$A_X(\mathcal{Q}) = A_X(u_1, \ldots, u_r) + A_X(v_1, \ldots, v_r), \quad A_P(\mathcal{Q}) = A_P(l_1, \ldots, l_N),$$
$$M_X(\mathcal{Q}) = M_X(u_1, \ldots, u_r) + M_X(v_1, \ldots, v_r), \quad M_P(\mathcal{Q}) = M_P(l_1, \ldots, l_N),$$

where $P = \{p_1, \ldots, p_r\}$. We set

$$A(\mathcal{Q}) = A_X(\mathcal{Q}) + A_P(\mathcal{Q}),$$
$$M(\mathcal{Q}) = M_X(\mathcal{Q}) + M_P(\mathcal{Q}),$$
$$L(\mathcal{Q}) = r.$$

Thus $A(\mathcal{Q})$ is the number of additions in $\mathcal{Q}$ and $M(\mathcal{Q})$ is the number of multiplications by constants in $\mathcal{Q}$ when each u_i, v_i, l_i is a linear formula in expanded form.

Given the quadratic forms $f_1, \ldots, f_N$, we define the cubic form

$$f = f_1 z_1 + \cdots + f_N z_N, \tag{3}$$

where $z_1, \ldots, z_N$ are new indeterminates. (If $f_1, \ldots, f_N$ are bilinear forms then f is a trilinear form; see the comment after the proof of the next lemma.)

LEMMA 2.1. *With the preceding notation* $||f|| \leq 2A(\mathcal{Q}) + 4L(\mathcal{Q}) + 2N - 1$, *where* $\mathcal{Q}$ *is any quadratic algorithm that computes* $f_1, \ldots, f_N$.

Proof. Let $r = L(\mathcal{Q})$. We build a formula for f from $\mathcal{Q}$ by writing

$$(f_1, \ldots, f_N)^{\mathrm{T}} = B(p_1, \ldots, p_r)^{\mathrm{T}},$$

where $B = (\beta_{ij})$ is an $N \times r$ matrix of scalars determined by the l_i in $\mathcal{Q}$. Set

$$(w_1, \ldots, w_r) = (z_1, \ldots, z_N)B.$$

Then

$$f = u_1 v_1 w_1 + \cdots + u_r v_r w_r,$$

and the right-hand side yields a formula F for f with

$$\begin{aligned} ||F|| = {} & A_X(u_1, \ldots, u_r) + M_X(u_1, \ldots, u_r) \\ & + A_X(v_1, \ldots, v_r) + M_X(v_1, \ldots, v_r) \\ & + A_Z(w_1, \ldots, w_r) + M_Z(w_1, \ldots, w_r) \\ & + 3r - 1. \end{aligned}$$

Now

$$A_Z(w_i) = \#(\text{nonzero } \beta_{1i}, \ldots, \beta_{Ni}) - 1.$$

(This assumes that no column of B is zero, i.e., each p_i in $\mathcal{Q}$ is actually used. Clearly, we may assume that this is the case.) Thus

$$\begin{aligned} A_Z(w_1, \ldots, w_r) &= \#(\text{nonzero } \beta_{ij}) - r \\ &= A_P(\mathcal{Q}) + N - r. \end{aligned}$$

Also,

$$M_Z(w_1, \ldots, w_r) = \#(\beta_{ij} \text{ different from } 0, \pm 1) = M_P(\mathcal{Q}).$$

Thus

$$||F|| = A(\mathcal{Q}) + M(\mathcal{Q}) + N + 2r - 1.$$

Finally,

$$\begin{aligned} ||F|| &\leq A(\mathcal{Q}) + M(\mathcal{Q}) + N + 2r - 1 \\ &\leq 2A(\mathcal{Q}) + 2N + 4r - 1. \end{aligned}$$

□

Suppose now that $f_1, \ldots, f_N$ are bilinear forms in the indeterminates $\{x_1, x_2, \ldots\}$, $\{y_1, y_2, \ldots\}$ and we restrict attention to bilinear algorithms. Then, as is well known, if we rewrite f as $g_1x_1 + g_2x_2 + \cdots$ or $h_1y_1 + h_2y_2 + \cdots$, we obtain two other dual sets of bilinear forms, $\{g_1, g_2, \ldots\}$ and $\{h_1, h_2, \ldots\}$, whose rank is the same as that of $f_1, \ldots, f_N$ (e.g., see Pan [6], Hopcroft and Musinski [3], or Brockett and Dobkin [2]). This gives us two other ways in which $||f||$ can be estimated. (Note that the scalar complexity of the three sets of forms can be quite different; see [1].)

The remaining definitions are from Kalorkoti [4] but are considerably simplified because of the absence of division. Let Y, Z be finite sets of indeterminates over k which are disjoint from X. Suppose $f \in k[X, Y]$ and $g \in k[X, Z]$. We say that f *represents* g with respect to X if there is a map $\sigma : X \cup Y \to k[X, Z]$ with $\sigma(x) = x$ for all $x \in X$ and $\sigma(y) \in k[Z]$ for all $y \in Y$ such that $f^\sigma = g$ (here f^σ denotes the image of f under the ring homomorphism induced by σ). Set

$$f = \sum_{i=0}^{d} f_i(Y)M_i(X), \qquad g = \sum_{i=0}^{d} g_i(Z)M_i(X),$$

where for each i we have $f_i(Y) \in k[Y]$, $g_i(Z) \in k[Z]$ and the $M_i(X)$ are distinct power products in the indeterminates of X. Clearly, $f^\sigma = g$ if and only if $f_i^\sigma = g_i$ for each i so that, using $|Y|$ to denote the cardinality of Y,

$$\begin{aligned} |Y| &\geq \#(f_0, \ldots, f_d \text{ that are algebraically independent over } k) \\ &\geq \#(g_0, \ldots, g_d \text{ that are algebraically independent over } k) \end{aligned}$$

(see Zariski and Samuel [11] for material on algebraic independence). Now if we define

$$\mathrm{td}_X(g) = \#(g_0, \ldots, g_d \text{ that are algebraically independent over } k)$$

(td stands for transcendence degree), then we have the following lemma.

LEMMA 2.2. *If f represents g with respect to X then $|Y| \geq \mathrm{td}_X(g)$.*

The main result is an algebraic analogue to Nečiporuk's [5] lower bound for the formula size of boolean functions.

THEOREM 2.3. *Suppose $f \in k[X]$ and $f \notin k[X']$ for any $X' \subset X$. Let $X_1, \ldots, X_t$ be a partition of X into pairwise disjoint nonempty sets. Then*

$$||f|| \geq \frac{1}{2} \sum_{i=1}^{t} \mathrm{td}_{X_i}(f) - 1.$$

Proof. The proof is virtually identical to the one given by Savage [7, pp. 101–103] with the preceding lemma playing the role of the number of "subfunctions" of a boolean function. The boolean function $\eta \cdot x \oplus \gamma$ is replaced by $z_1x + z_2$, where z_1, z_2 are a new pair of indeterminates for each use of the function. The "free constants" of [7] are replaced by new indeterminates.

Our formula (which computes f) starts out as a binary tree, and after the transformations we produce a new formula (with a slightly extended set of operations) whose result represents f with respect to the set X_i, whose indeterminates play the part of the "variables" for the transformation.

We now give details of the proof for the sake of readers who are unfamiliar with [7]. Let F be an optimal formula for f. Let l_i denote the number of leaves of F labelled by elements of X_i, where $1 \leq i \leq t$. Then

$$||F|| \geq \sum_{j=1}^{t} l_i - 1. \tag{4}$$

Now fix i and change F as follows:

(a) If a subtree has leaves labelled only from $(X - X_i) \cup k$ then replace it by a single new leaf labelled by a new indeterminate.

(b) If a subtree has only one leaf labelled by an indeterminate $x \in X_i$ then replace it by a new special computation vertex with three inputs z_1, z_2, x which computes $z_1 x + z_2$, where z_1, z_2 are new indeterminates. (This extends the notion of formula to allow for the special vertices; in fact we could just implement the computation of $z_1 x + z_2$ by a normal formula but this doesn't improve the result.)

Let F_i be the formula obtained after all the transformations and let Z_i be the set of new indeterminates introduced. The result of F_i represents f with respect to X_i so that, by Lemma 2.2,

$$|Z_i| \geq \mathrm{td}_{X_i}(f). \tag{5}$$

Now let d_1, d_2 denote the number of internal vertices that are roots of subtrees of F_i with one and more than one leaf labelled by an element of X_i, respectively. Then it is easy to see that $d_2 \leq l_i - 1$ and $d_1 \leq l_2$ (for the first of these inequalities we need $l_i > 0$ and this follows from the assumption that $f \notin k[X']$ for any $X' \subset X$). Thus

$$||F_i|| = d_1 + d_2 \leq 2l_i - 1. \tag{6}$$

If we ignore those leaves of F_i that are attached to special computation vertices and labelled by an indeterminate from X_i then we have a binary tree with at least $|Z_i|$ leaves so that $||F_i|| \geq |Z_i| - 1$. Using the last inequality together with (6) we have $l_i \geq |Z_i|/2$. The result now follows from (4) and (5). □

The growth rate of the sum in the theorem is bounded by $|X|^2$ since $\mathrm{td}_{X_i}(f) \leq |X| - |X_i|$. Moreover, the constant factor is best possible. Consider the inner product

$$f_n = x_1 y_1 + \cdots + x_n y_n.$$

Clearly, $||f_n|| \leq 2n - 1$. Suppose first that $n > 1$. Then, using the partition $X_1 = \{x_1\}, \ldots, X_n = \{x_n\}, X_{n+1} = \{y_1\}, \ldots, X_{2n} = \{y_n\}$ we have $\mathrm{td}_{X_i}(f) = 2$, and so the theorem yields $||f_n|| \geq 2n - 1$. If $n = 1$ the theorem yields the trivial lower bound 0. However, $||f_2|| \leq 2||f_1|| + 1$ and since the theorem yields $||f_2|| \geq 3$ we deduce that $||f_1|| \geq 1$. The observation used here is that f_2 is the sum of two disjoint copies of f_1. In general we can try to gain a small improvement to the bound given by the theorem by replacing the polynomial by the sum of two disjoint copies of it (we will do this in Theorem 3.1).

3. Matrix multiplication. We now revert to the notation of §1. The associated trilinear form is

$$t_{mnp} = \sum_{i=1}^{m} \sum_{j=1}^{p} w_{ij} z_{ij}, \tag{7}$$

where the z_{ij} are new indeterminates.

THEOREM 3.1. $||t_{mnp}|| \geq mnp/2 + mp + n/2 - 1$.

Proof. Introduce new sets of indeterminates $\hat{X}$, $\hat{Y}$, $\hat{Z}$ whose elements are in one-to-one correspondence with those of X, Y, Z, respectively ($x_{ij} \leftrightarrow \hat{x}_{ij}$, etc.). Let $T_{mnp} = t_{mnp} + \hat{t}_{mnp}$ and note that $||T_{mnp}|| \leq 2||t_{mnp}|| + 1$, which yields

$$||t_{mnp}|| \geq (||T_{mnp}|| - 1)/2. \tag{8}$$

Consider the following partition of the indeterminates:

$$P_s = \{x_{1s}, x_{2s}, \ldots, x_{ms}, y_{s1}, y_{s2}, \ldots, y_{sp}\}, \quad 1 \leq s \leq n,$$
$$Q_{ij} = \{z_{ij}\}, \quad 1 \leq i \leq m, \quad 1 \leq j \leq p$$

with $\hat{P}_s$ and $\hat{Q}_{ij}$ defined similarly. We claim that $\mathrm{td}_{P_s}(T_{mnp}) \geq mp + 1$ for each s. To see this note that the coefficient of $x_{is}y_{sj}$ in T_{mnp} is z_{ij}, while the coefficient of 1 is $\hat{t}_{mnp}$. The claim now follows. Similarly, $\mathrm{td}_{\hat{P}_s}(T_{mnp}) \geq mp + 1$.

We also claim that $\mathrm{td}_{Q_{ij}}(T_{mnp}) = 2$, because we have $T_{mnp} = w_{ij}z_{ij} + r_{ij} + \hat{t}_{mnp}$, where

$$r_{ij} = \sum_{\substack{1 \leq u \leq m \\ 1 \leq v \leq p \\ (u,v) \neq (i,j)}} w_{uv}z_{uv}.$$

Clearly, w_{ij} and $r_{ij} + \hat{t}_{mnp}$ are algebraically independent over k, so the claim follows. Similarly, $\mathrm{td}_{\hat{Q}_{ij}}(T_{mnp}) = 2$.

The lower bound now follows from Theorem 2.3 and (8). □

Note that by symmetrical arguments we can show that $||t_{mnp}|| \geq mnp/2 + mn + p/2 - 1$ and $||t_{mnp}|| \geq mnp/2 + np + m/2 - 1$. Thus

$$||t_{mnp}|| \geq mnp/2 + \max(mp + n/2, mn + p/2, np + m/2) - 1.$$

Furthermore, the formula implicit in (7) shows that

$$||t_{mnp}|| \leq 2mnp + mp - 1,$$

which can be improved to

$$||t_{mnp}|| \leq 2mnp + \min(mn, mp, pn) - 1$$

simply by rebracketing (7). We therefore have a good estimate of $||t_{mnp}||$. Combining the lower bound with Lemma 2.1, we see that

$$A(\mathcal{Q}) + L(\mathcal{Q}) \geq (mnp - \min(2mp - n, 2mn - p, 2np - m))/8$$

for any quadratic algorithm $\mathcal{Q}$ that computes XY.

Taking $m = p = n$ so that we are looking at square matrices, we have

$$A(\mathcal{Q}) + L(\mathcal{Q}) \geq n^3/8 - n^2/4 + n/8,$$

where $\mathcal{Q}$ is any quadratic algorithm for $n \times n$ matrix multiplication. Thus any $o(n^3)$ quadratic algorithm for this problem must reuse additions to a great extent.

Acknowledgment. I thank an anonymous referee whose helpful comments led to improvements in the paper.

REFERENCES

[1] A. BORODIN AND I. MUNRO, *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, New York, 1975.

[2] R. W. BROCKETT AND D. DOBKIN, *On the optimal evaluation of a set of bilinear forms*, Linear Algebra Appl., 19 (1978), pp. 207-235.

[3] J. E. HOPCROFT AND J. MUSINSKI, *Duality applied to the complexity of matrix multiplication and other bilinear forms*, SIAM J. Comput., 2 (1973), pp. 188-194.

[4] K. KALORKOTI, *A lower bound for the formula size of rational functions*, SIAM J. Comput., 14 (1985), pp. 678-687.

[5] È. I. NEČIPORUK, *A boolean function*, Dokl. Akad. Nauk SSSR, 169 (1966), pp. 765-766; Soviet Math. Dokl., 7 (1966), pp. 999-1000.

[6] V. Y. PAN, *On schemes for the evaluation of products and inverses of matrices*, Uspekhi Mat. Nauk, 27 (1972), pp. 249-250. (In Russian.)

[7] J. E. SAVAGE, *The Complexity of Computing*, John Wiley, New York, 1976.

[8] V. STRASSEN, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354-356.

[9] ———, *Vermeidung von Divisionen*, J. Reine Angew. Math., 264 (1973), pp. 184-202.

[10] S. WINOGRAD, *On the number of multiplications necessary to compute certain functions*, Comm. Pure Appl. Math., 23 (1970), pp. 165-179.

[11] O. ZARISKI AND P. SAMUEL, *Commutative Algebra*, Vol. I, Springer-Verlag, Berlin, 1986.

SIAM J. COMPUT.
Vol. 24, No. 6, pp. 1313-1331, December 1995

011

SCHEDULING PARALLEL MACHINES ON-LINE*

DAVID B. SHMOYS†, JOEL WEIN‡, AND DAVID P. WILLIAMSON§

Abstract. The problem of scheduling jobs on parallel machines is studied when (1) the existence of a job is not known until its unknown release date and (2) the processing requirement of a job is not known until the job is processed to completion. Two general algorithmic techniques are demonstrated for converting existing polynomial-time algorithms that require complete knowledge about the input data into algorithms that need less advance knowledge. Information-theoretic lower bounds on the length of on-line schedules are proven for several basic parallel machine models, and almost all of our algorithms construct schedules with lengths that either match or come within a constant factor of the lower bound.

Key words. scheduling, approximation algorithms

AMS subject classifications. 68A10, 68Q25, 90B35, 68R99

1. Introduction.

The scheduling of a set of tasks on parallel machines is a basic problem in combinatorial optimization with a number of increasingly important applications. There is a rich literature on parallel machine scheduling and on deterministic scheduling in general, but the overwhelming majority of these results assume that a complete specification of the instance is available before the algorithm begins to construct a schedule. This fails to capture many scheduling problems that arise in practice. For example, consider the allocation of jobs to the processing units of a multiprocessor: the scheduler does not have complete knowledge of a job's running time in advance or of what jobs will be created and require processing in the future. In this paper we will study on-line algorithms, algorithms that work without any clairvoyant assumptions, for the most basic types of parallel machine models. Our algorithms are based on two rather general techniques that allow us to convert algorithms that need more complete knowledge of the input data into ones that need less advance knowledge.

When on-line scheduling was studied in the past, the models that were considered were typically of the following form: the existence of a job is unknown until a certain *release date*, at which point the processing requirement for that job is completely specified. We will consider more realistic models, where the processing requirement of a job is also unknown when it starts processing and can only be determined by processing the job and observing how long it takes to be completed. In fact, our results show that the traditional sort of on-line scheduling problem is provably not much harder than its off-line analogue, whereas the lack of knowledge about job sizes can drastically affect the quality of solutions that can be obtained.

*Received by the editors May 4, 1993; accepted for publication (in revised form) July 11, 1994. This research was partially supported by NSF Presidental Young Investigator award CCR-89-96272 with matching support from United Parcel Service, Sun, the National Science Foundation, the Air Force Office of Scientific Research, and the Office of Naval Research, through NSF grant DMS89-20550 and Defense Advanced Research Projects Agency contract N00014-89-J-1988. A preliminary version of this paper appeared in the Proceedings of the 32nd Symposium on Foundations of Computer Science, October 1991. A short statement of preliminary results appeared in the Proceedings of the Center for Discrete Mathematics and Theoretical Computer Sciences Workshop on On-line Algorithms.

†School of Operations Research and Industrial Engineering, Cornell University, Ithaca, NY 14853 (shmoys@cs.cornell.edu).

‡Department of Computer Science, Polytechnic University, Five MetroTech Center, Brooklyn, NY 11201 (wein@mem.poly.edu). Additional support for this author was provided by an Army Research Office graduate fellowship, NSF grant CCR-9211494, and a grant from the New York State Science and Technology Foundation, Center for Advanced Technology in Telecommunications.

§IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598 (dpw@watson.ibm.com). Additional support for this author was provided by an NSF graduate fellowship, Defense Advanced Research Projects Agency contracts N00014-89-J-1988 and N00014-87-K-825, and Air Force Contract AFOSR-89-0271.

We will study the three basic types of parallel machine models [20]. In each, there are n jobs to be scheduled on m machines. Each machine can process at most one job at a time, and each job must be processed in an uninterrupted fashion on one of the machines. In the most general setting, the machines are *unrelated*: job j takes $p_{ij} = p_j/s_{ij}$ time units when processed by machine i, where p_j is the *processing requirement* (or *size*) of job j and s_{ij} is the speed of machine i for job j. If the machines are *uniformly related*, then each machine i runs at a given speed s_i for all jobs j and the processing time p_{ij} is given by p_j/s_i. Finally, for *identical* machines, we assume that $s_i = 1$ for each machine i. Let $p_{\max} = \max_j p_j$. If C_j denotes the time at which job j completes processing in a schedule, then the *makespan* or *length* of the schedule is $C_{\max} = \max_j C_j$. For a given instance $\mathcal{I}$, our objective is to find a schedule of minimum length $C^*_{\max}(\mathcal{I})$.

In an off-line setting, these three types of parallel machine models have been studied extensively. The associated scheduling problems are all strongly $\mathcal{NP}$-hard [17], and polynomial approximation schemes are known when the machines are either identical or uniformly related [23], [24]. For unrelated machines, obtaining a solution better than $(3/2)C^*_{\max}$ is $\mathcal{NP}$-hard, whereas a schedule of length at most $2C^*_{\max}$ can be found in polynomial time [31]. We will also consider the *preemptive* versions of these models, in which a job may be interrupted on one machine and continued later (possibly on another machine) without penalty. In each of these three models, there is a polynomial-time off-line algorithm for finding an optimal preemptive solution [25], [30], [33].

We shall evaluate on-line algorithms in terms of their *competitive ratio* [39]. Let $C^{\mathcal{A}}_{\max}(\mathcal{I})$ be the makespan of a deterministic on-line algorithm $\mathcal{A}$ on instance $\mathcal{I}$. Algorithm $\mathcal{A}$ is said to have *competitive ratio* c (or is said to be *c-competitive*) if $C^{\mathcal{A}}_{\max}(\mathcal{I}) \leq c \cdot C^*_{\max}(\mathcal{I}) + O(1)$ for all problem instances $\mathcal{I}$. If $\mathcal{A}$ is a randomized algorithm, then A is said to have competitive ratio c (or is said to be c-competitive) if $E[C^{\mathcal{A}}_{\max}(\mathcal{I})] \leq c \cdot C^*_{\max}(\mathcal{I}) + O(1)$ for all instances $\mathcal{I}$, where the expectation is taken over all random choices of the algorithm $\mathcal{A}$. Although these notions apply to algorithms without any restrictions on their running times, we will focus on polynomial-time on-line algorithms rather than the purely information-theoretic analogue. Nonetheless, our lower bounds are based on information-theoretic arguments.

In a nonpreemptive model, it may be unrealistic to assume that once a job is started it must be run until its (unknown) completion time without any form of recourse. A central aspect of our models is that we introduce the notion of *restarts*: a job may be canceled and later started again, but it is started again from scratch. For example, in the uniformly related machine model, we may wish to cancel a job that is taking longer than "anticipated," and then start it again on a faster machine.

The results of this paper are as follows. We introduce two general techniques to convert off-line algorithms into algorithms that require less initial information. Using the first technique, we show that we can focus on the case when all jobs are available at time 0, since the situation in which there are unknown job arrivals and unknown processing times can be reduced, with only a factor of 2 increase in the competitive ratio, to one in which only the processing times are unknown. This result also holds when comparing a model in which only the arrivals are unknown to its off-line equivalent. As a consequence, we consider the situation in which all jobs (of unknown size) are available to be scheduled from the start. For both uniformly related and unrelated machines, we use our second technique to convert off-line algorithms into algorithms that need not be given the processing time of each job. Nonetheless, the resulting on-line algorithms do not suffer too great a degradation in the quality of the solution produced.

It is quite simple to obtain tight bounds for the identical machine case: one of the oldest results in scheduling theory is an on-line algorithm of Graham [19], which produces a schedule

of length at most $(2 - \frac{1}{m})C^*_{\max}$; we give a straightforward proof that this is exactly the best possible ratio. We also give an identical tight bound on the competitive ratio obtainable in the preemptive variant. This has the important consequence that, although complexity theory shows that there is a fundamental difference between the preemptive and nonpreemptive models, this difference disappears when scheduling jobs on-line. We also show that randomization is of little help to the scheduler, proving that no randomized algorithm can achieve competitive ratio better than $(2 - O(\frac{1}{\sqrt{m}}))$, even against an oblivious adversary. This result is in sharp contrast to other recent work in on-line algorithms, in which randomization has been shown to significantly increase the performance of the algorithms [29], [40].

We then show that on-line scheduling on uniformly related machines is much harder than on identical machines. This is also quite different from the off-line setting, where results for identical machines have typically extended to the case where machines run at different speeds. In our on-line model, we show that this generalization does make the problem significantly harder: we prove that the optimal competitive ratio is $\Theta(\log m)$. We generalize this model to unrelated machines by assuming that for each job the relative speeds of the machines are known, but its size is unknown. In this setting, we can also obtain an on-line algorithm with an $O(\log n)$ competitive ratio; this upper bound is only tight if n is polynomial in m, which need not be the case. Once again, we also give identical results for the preemptive variants of these models. For uniformly related machines, we also show how to take advantage of the situation that the relative speeds of the machines are not too different and give an $O(\log R)$-competitive algorithm, where R is the ratio of the fastest-to-slowest machine speeds. Finally, we can show that this bound is tight in the following sense: we prove a lower bound of $\Omega(\log R)$ on the competitive ratio of any deterministic on-line scheduling algorithm for the scenario in which the ratio of fastest-to-slowest machine speeds is equal to R, $R < m$.

On-line algorithms have been studied for a variety of problem domains. Some of the oldest of these results are for the bin-packing problem. When the number of bins is fixed, on-line bin-packing can be interpreted as a type of on-line scheduling, where the jobs are given in a list and scheduled in turn. The job currently being scheduled is completely specified, but the jobs later in the list are completely unknown. Faigle, Kern, and Turan [12] have proved some lower bounds in this model, and several authors have given improved algorithms [5], [16], [35]. This model, however, is rather different from the ones we consider. There are many recent results on on-line algorithms for problem domains that range from classic problems in combinatorial optimization [26], [29], [40], to various problems in data and memory management [28], [39], to the k-server problem [15], [32].

In terms of previous work on our model of on-line scheduling, some attention has been given in the past to the question of unknown release dates. In the preemptive model, Gonzales and Johnson gave a polynomial-time algorithm that optimally solves this problem on identical machines [18]. Sahni and Cho extended this result to apply to uniformly related machines [37]. In the nonpreemptive model, Gusfield considered a more general problem on identical machines, in which each job has an associated due date and the goal is to minimize the maximum lateness. He proved a bound of $(2 - \frac{1}{m})p_{\max}$ on the difference between the maximum lateness produced by an on-line heuristic and the minimum possible maximum lateness [21].

Relatively little work has been done on scheduling problems with unknown job sizes. Chandra, Karloff, and Vishwanathan [8] proposed studying on-line scheduling with unknown processing times and analyzed the problem of minimizing the average completion time on a single machine with preemption. In addition to the algorithms for identical machines given by Graham [19], the only other work for parallel machines known to us prior to ours is that of Jaffe [27] and Davis and Jaffe [10]. Davis and Jaffe show that in a restricted model without

restarts, any on-line algorithm for nonpreemptive scheduling of uniformly related machines has competitive ratio $\Omega(\sqrt{m})$. Jaffe gives an algorithm for this case with competitive ratio $O(\sqrt{m})$.

Subsequent to our work there have been a number of papers considering variants of on-line scheduling. Feldmann, Sgall, and Teng [14] studied on-line scheduling with unknown job sizes on a mesh of identical processors, where one must allocate a submesh of a specified size to a job when the processing time of that job is unknown. They prove a $\Theta(\sqrt{\log \log m})$ bound on the competitive ratio in this model. In addition, they study a number of other architectures such as hypercubes and trees. In a subsequent paper they, along with Kao, extend their results to the important scenario in which there are precedence constraints between the jobs [13].

Motwani, Phillips, and Torng considered a number of problems relating to on-line preemptive scheduling [34]. Deng and Koutsoupias [11] considered on-line scheduling of jobs with precedence constraints and communication delays, in which both the job sizes and the structure of the precedence constraint dag are unknown. Awerbuch, Kutten, and Peleg [2] studied distributed versions of on-line scheduling. In addition, recently there have been several interesting papers that considered the closely related problem of *on-line load balancing* in various parallel machine models [1], [3], [4].

The rest of this paper is organized as follows. In §2 we show that the introduction of unknown release dates into a scheduling problem does not make the problem too much harder. As a result, we concentrate on the situation where all the jobs are available at time 0 but have unknown processing requirements. In §3 we present our on-line scheduling algorithms for the various parallel machine models, and in §4 we give the corresponding lower bounds.

Note that throughout this paper all logarithms are taken base 2.

2. Unknown release dates. Our model of on-line scheduling includes both unknown release dates for jobs and unknown job sizes. In this section we will show that, with respect to minimizing schedule length, the first element has a relatively small impact.

We will show that in a scheduling environment with unknown release dates, we can construct a schedule by repeatedly using an algorithm $\mathcal{A}$ which works in the simpler environment in which all jobs are available at time 0. The quality of the schedule thus constructed is within a factor of 2 of the quality of schedules constructed by $\mathcal{A}$ in the simpler environment. This result does not depend on the remaining specifics of the scheduling environment; in particular, it allows us to use off-line algorithms to obtain algorithms that can handle unknown release dates (but where the processing times are known once released), as well as allowing us to focus on on-line algorithms in the case when all jobs are released at time 0.

THEOREM 2.1. *Let $\mathcal{A}$ be a polynomial-time scheduling algorithm that works in an environment in which each job to be scheduled is available at time 0 and which always produces a schedule of length at most $\rho C^*_{\max}$. For the analogous environment in which the existence of a job is unknown until its release date, there exists another polynomial-time algorithm $\mathcal{A}'$ that works in this more general setting and produces a schedule of length at most $2\rho C^*_{\max}$.*

Proof. Let $\mathcal{I}$ be an instance with jobs of unknown release dates and let S_0 be the set of jobs available at time 0. The scheduler applies algorithm $\mathcal{A}$ and schedules the jobs in S_0, finishing at time F_0. Let S_1 be the set of jobs released in the time interval $(0, F_0]$. The scheduler now, at time F_0, applies algorithm $\mathcal{A}$ to schedule S_1, finishing at time F_1. In general, let S_{i+1} be the set of jobs released in $(F_{i-1}, F_i]$ and let F_i be the point in time when the schedule for S_i completes. At time F_i, the scheduler uses algorithm $\mathcal{A}$ to schedule the jobs in S_{i+1}. If $S_{i+1} = \emptyset$, then all machines remain idle until the time t when the next job is released; we then let $F_{i+1} = t$. Let F_k be the finishing point of the entire schedule. (Figure 1 shows the structure of the schedule constructed by the algorithm.)

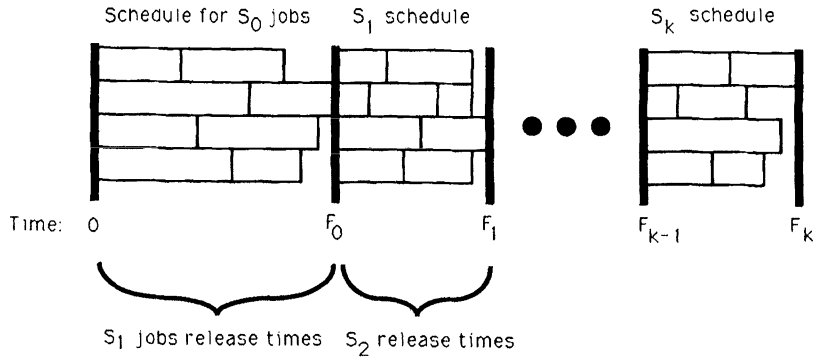


FIG. 1. *Using an algorithm for a scheduling environment without release dates to schedule in an environment with release dates.*

To analyze the length of the resulting schedule, consider the last instance $\mathcal{I}'$ on which the algorithm $\mathcal{A}$ is executed, that is, the set of jobs in S_k with their release dates modified to be 0. Algorithm $\mathcal{A}$ produces a schedule of length at most $\rho C^*_{\max}(\mathcal{I}')$; this means that $F_k - F_{k-1} \leq \rho C^*_{\max}(\mathcal{I}') \leq \rho C^*_{\max}(\mathcal{I})$.

Consider the instance $\mathcal{I}''$ obtained from $\mathcal{I}'$ by modifying the release dates to F_{k-2}. Since each job in $\mathcal{I}''$ is released earlier than its corresponding job in $\mathcal{I}$, $C^*_{\max}(\mathcal{I}) \geq C^*_{\max}(\mathcal{I}'')$. Furthermore, we know that $C^*_{\max}(\mathcal{I}'') = F_{k-2} + C^*_{\max}(\mathcal{I}')$. Putting the pieces together, we see that

$$C^*_{\max}(\mathcal{I}) \geq F_{k-2} + C^*_{\max}(\mathcal{I}') \geq F_{k-2} + (F_k - F_{k-1})/\rho.$$

Similarly, $F_{k-1} - F_{k-2} \leq \rho C^*_{\max}(\mathcal{I})$. (If $S_{k-1} = \emptyset$, we have the stronger inequality, $F_{k-1} \leq C^*_{\max}(\mathcal{I})$.) By combining these inequalities, we get that

$$F_k \leq \rho C^*_{\max}(\mathcal{I}) + F_{k-1} - \rho F_{k-2} \leq \rho C^*_{\max}(\mathcal{I}) + F_{k-1} - F_{k-2} \leq 2\rho C^*_{\max}(\mathcal{I}). \qquad \square$$

This theorem is very general in that it can be applied to a number of different scheduling environments. In particular, it shows that to produce an algorithm for our full on-line model, we can modify an algorithm for the case in which all jobs are available at time 0 and processing times are unknown, thereby increasing the competitive ratio of the algorithm by only a factor of 2. Furthermore, the theorem applies not only to problems of parallel machine scheduling but also to the entire class of shop scheduling problems, including open shop, flow shop, and job shop [38]. In addition, it applies to the scheduling model of Feldmann, Sgall, and Teng [14]. They studied the on-line allocation of submeshes of a large mesh to different jobs, but their algorithms only worked when all jobs were available at time 0. Our theorem generalizes their result to a $\Theta(\sqrt{\log \log m})$ on-line algorithm even when jobs have unknown release dates. It also generalizes their results on other architectures as well.

Finally, our theorem yields the following corollary.

COROLLARY 2.2. *If job release dates are unknown, but once a job arrives its size is known, there is a polynomial-time* on-line *algorithm for scheduling uniformly related machines with competitive ratio* $(2 + \epsilon)$, *for any fixed* ϵ, *and a polynomial-time* on-line *algorithm for scheduling unrelated machines with competitive ratio* 4.

Proof. This follows directly from Theorem 2.1 and previously known results on the off-line approximability of these problems [24], [31]. $\square$

For identical machines this result yields a $(2+\epsilon)$-approximation algorithm; however, something slightly better was already known. In 1966 Graham showed that list scheduling was a $(2-\frac{1}{m})$-approximation algorithm for scheduling identical machines. In list scheduling, the scheduler takes any list of jobs and, whenever a machine becomes available, places the next job on the list on that machine. It is not hard to see that if this strategy is extended so that newly arriving jobs are added to the end of the list, then list scheduling is a $(2-\frac{1}{m})$-approximation algorithm for scheduling identical machines with unknown release dates (see, for example, [21], [22]).

Despite the fact that unknown release dates do not make a scheduling problem *much* more difficult, we can show that they sometimes do make it more difficult to schedule machines near-optimally.

THEOREM 2.3. *Let $\mathcal{A}$ be a deterministic on-line algorithm for nonpreemptive scheduling of identical machines with unknown release dates but known processing requirements. Then the competitive ratio of algorithm $\mathcal{A}$ is at least $\frac{10}{9}$, even if restarts are allowed.*

It is interesting to note that preemption makes a difference in this case; Gonzales and Johnson have shown that there is a polynomial-time algorithm for solving the problem optimally when preemption is allowed [18]. Intuitively this is not surprising, since preemption allows the algorithm to adjust to new information without losing work done beforehand. For the sake of coherence of presentation, we defer the proof of this theorem to §4.2.

In light of the results in this section, for the remainder of this paper, except §4.2, we shall focus on scheduling environments in which all jobs are available to be scheduled at time 0.

3. Algorithms for on-line scheduling. In this section we will present on-line scheduling algorithms for the basic parallel machine models. We first note that in the case of identical machines, the well-known list scheduling algorithm of Graham [19] always comes within a factor of $(2-\frac{1}{m})$ of the optimal length schedule and comes within the same bound of the optimal preemptive schedule length. Since list scheduling does not depend on the sizes of the jobs, list scheduling is an on-line algorithm with a $(2-\frac{1}{m})$ competitive ratio.

THEOREM 3.1 (Graham). *There is an on-line algorithm for scheduling identical machines that achieves competitive ratio $(2-\frac{1}{m})$ in both the preemptive and nonpreemptive models.*

For the other machine models, we will present a general technique which yields an $O(\log n)$-competitive algorithm for each of them, assuming the machine speeds are known to the scheduler. We will then show how to convert this general algorithm to an $O(\min(\log m, \log(s_1/s_m)))$-competitive algorithm for both preemptive and nonpreemptive uniformly related machines, where $s_1 \geq s_2 \geq \cdots \geq s_m$. We will also present a different algorithm for nonpreemptive uniformly related machines; although its competitiveness is no better than our general technique, it makes use of a new and interesting relaxed decision procedure for the scheduling of uniformly related machines. In addition, it was applied by Aspnes et. al. with essentially no modification to yield a theorem about on-line load balancing on uniformly related machines [1].

3.1. The general technique. Our general algorithm depends on the existence of either polynomial-time optimization algorithms or polynomial-time *ρ-approximation algorithms* for scheduling in the various machine models. We define a ρ-approximation algorithm as a polynomial-time algorithm that always produces a solution with makespan no more than a factor of ρ larger than optimal.

THEOREM 3.2. *Let $\mathcal{A}$ be an off-line ρ-approximation algorithm for the (nonpreemptive/preemptive) (uniformly related/unrelated) parallel machine scheduling problem. Let $\mathcal{I}$ be an instance of this problem. Then there is an on-line scheduling algorithm that, for any instance $\mathcal{I}$, produces a schedule of length at most $(4\rho \log n + 4\rho \log 2\rho + 1)C^*_{\max}(\mathcal{I})$.*

Proof. The on-line algorithm works by repeatedly applying algorithm $\mathcal{A}$ to the remaining unscheduled jobs after guessing the size of each job. Given a schedule produced by the algorithm $\mathcal{A}$, our on-line algorithm will run each job at the particular time interval and on the particular machine specified by the schedule. In the preemptive model, the job might not have finished all of its processing by the end of the time allotted to it, in which case we preempt the job. In the nonpreemptive model, we cancel the job if it is not completely processed in the time allotted, and we will restart the job later. In either case, if the job does not complete, we will be able to update our estimate of the size of that job.

For the sake of simplicity, we will assume that the data is normalized so that the fastest machine for each job j has speed $s_{ij} = 1$. One result of this assumption is that any job of size p_j takes time p_j to complete on the machine that processes it fastest.

The complete on-line algorithm is below.

MAIN ALGORITHM.

Step 1. Pick any job j' and run it to completion on a machine $m_{i'}$ such that $s_{i',j'} = 1$. Let the time that this takes be denoted by Δ.

Step 2. Let $q = \Delta/\rho n$.

Step 3. Use algorithm $\mathcal{A}$ to construct a schedule for all jobs that have not yet completed, setting $p_j \leftarrow q$ for all remaining jobs j. Run the jobs in this schedule, preempting or canceling all jobs that do not complete in the time allotted to them by the schedule.

Step 4. If any job has not yet completed, set $q \leftarrow 2q$ and go to Step 3.

Let $C^*_{\max}$ be the length of the optimal schedule. We will now analyze the algorithm and the length of the schedule it produces. First, in Step 1, the time Δ taken by job j' on machine $m_{i'}$ is at most $C^*_{\max}$, since the optimal schedule can be no shorter than the time taken by any job running on the machine which processes it fastest.

Next, we show that the first iteration of Step 3 produces a schedule no longer than $\Delta \leq C^*_{\max}$. One way to construct a schedule is to assign each of the n jobs to the machine that processes it the fastest. In the worst case, all n jobs would be assigned to the same machine, and this schedule would have length $nq = \Delta/\rho$. Since the schedule produced by algorithm $\mathcal{A}$ is no longer than ρ times optimal, it must produce a schedule of length no longer than $\Delta \leq C^*_{\max}$.

In addition, future iterations of Step 3 must produce schedules of length at most $2\rho C^*_{\max}$. Suppose the algorithm performs an iteration of Step 3 in which the jobs are assigned size q. Since the algorithm did not finish processing each of the remaining jobs in the previous iteration, we know that each job j in the instance being scheduled is such that $p_j > q/2$. An optimal schedule for this subset of jobs must take time no greater than $C^*_{\max}$. We can transform this optimal schedule into a schedule for the instance in which each job is processed for q units so that the length of the schedule increases by no more than a factor of 2. Finally, the algorithm $\mathcal{A}$ will find a schedule for this instance that is no more than ρ times as long as the optimal schedule, and so the length of the schedule found by algorithm $\mathcal{A}$ is at most $2\rho C^*_{\max}$.

To derive our $O((\rho \log n)C^*_{\max})$ bound on the length of the complete schedule, we show that we need to consider essentially only the last $\log(2\rho n)$ iterations of Step 3.

LEMMA 3.3. *Let f denote the number of iterations of Step 3. Then the length of the schedule produced in iteration $f - i$ is at least 2^ℓ times as long as the length of the schedule produced by algorithm $\mathcal{A}$ in iteration $f - i - \ell \log(2\rho n)$.*

Proof. Assume that the (estimated) job size in iteration $f - i$ is q; then the (estimated) job size in iteration $f - i - \ell \log(2\rho n)$ is $q/(2\rho n)^\ell$. If a job of size q exists, then the schedule containing it must take time at least q. As we showed earlier, a schedule produced by algorithm $\mathcal{A}$ for jobs with size $q/(2\rho n)^\ell$ has length at most $\rho nq/(2\rho n)^\ell$. Thus the length of the schedule

produced when the job size is $q/(2\rho n)^\ell$ is at most $(1/2)^\ell$ times the length of the schedule produced when the job size is q.

Since every $\log(2\rho n)$ iterations the length of the schedule produced doubles, we can "charge" iterations $f - i - \ell \log(2\rho n)$, $1 \leq \ell \leq (f-i)/\log(2\rho n)$, to iteration $f - i$. Since each of the last $\log(2\rho n)$ iterations has length no longer than $2\rho C^*_{\max}$, and each gets charged no more than $\frac{1}{2} + \frac{1}{4} + \cdots + (\frac{1}{2})^{\lceil f/\log 2\rho n \rceil}$ times its length, the overall length of the schedule is at most

$$\Delta + (\log 2\rho n)\left(2\rho C^*_{\max}\left(1 + \frac{1}{2} + \cdots + \left(\frac{1}{2}\right)^{\lceil \frac{f}{\log 2\rho n} \rceil}\right)\right)$$

$$\leq (4\rho \log n) C^*_{\max} + (4\rho \log 2\rho) C^*_{\max} + C^*_{\max}. \qquad \square$$

Since there exists a polynomial-time algorithm for finding an optimal preemptive schedule for unrelated machines due to Lawler and Labetoulle [30], and there exists a 2-approximation algorithm for scheduling nonpreemptive unrelated machines due to Lenstra, Shmoys, and Tardos [31], we have the following corollaries.

COROLLARY 3.4. *There is a (polynomial-time) on-line algorithm for scheduling preemptive unrelated machines that has competitive ratio* $4(\log n) + 5$.

COROLLARY 3.5. *There is a (polynomial-time) on-line algorithm for scheduling nonpreemptive unrelated machines that has competitive ratio* $8(\log n) + 17$.

We can do somewhat better with uniformly related machines. As the following lemma shows, by applying a list scheduling algorithm until there are at most m unfinished jobs, we can quite easily reduce the number of jobs from n to $m - 1$.

LEMMA 3.6. *The number of jobs in any uniformly related machine problem instance can be reduced on-line from* n *to* $m - 1$, *while increasing the competitive ratio by* 1.

Proof. We simply start by applying the list scheduling algorithm to the jobs. Assign initial jobs to the machines arbitrarily; whenever a job completes and a machine becomes idle, we assign an unprocessed job to it. When no jobs remain, at most $m - 1$ jobs have not yet finished processing. Furthermore, the length of the schedule to this point in time can be no greater than $\sum_j p_j / \sum_i s_i$, which is a lower bound on the length of the optimal preemptive and nonpreemptive schedules. $\square$

In the preemptive setting, we can use the algorithm of Horvath, Lam, and Sethi [25] to obtain an optimal solution in each iteration of Step 3. Although the general problem of scheduling nonpreemptively on uniformly related parallel machines is $\mathcal{NP}$-hard, the instances on which algorithm $\mathcal{A}$ must run are extremely simple, since each job has the same (estimated) processing time. An optimal solution of such an instance can be computed in $O(m \log m)$ time by a variant of list scheduling: assign the next job on the list to the machine on which it would complete earliest. Combining these algorithms with Theorem 3.2 and Lemma 3.6, we obtain the following corollary.

COROLLARY 3.7. *There is a (polynomial-time) on-line algorithm for scheduling (preemptive/nonpreemptive) uniformly related machines that has competitive ratio* $4(\log m) + 6$.

Let $1 = s_1 \geq s_2 \geq \cdots \geq s_m$ and let $R = 1/s_m$. If $R < m$, we can improve the results of Corollary 3.7 to $O(\log R)$ competitive algorithms.

LEMMA 3.8. *Consider the application of the main algorithm to an instance of preemptive or nonpreemptive scheduling of uniformly related parallel machines in which* $n \leq m$. *Let* f *denote the number of iterations of Step* 3. *Then the length of the schedule produced in iteration* $f - i$ *is at least* 2^ℓ *times as long as the length of the schedule produced in iteration* $f - i - \ell \log(2R)$.

Proof. Assume that the (estimated) job size in iteration $f - i$ is q; then the (estimated) job size in iteration $f - i - \ell \log(2R)$ is $q/(2R)^\ell$. If a job of size q exists, then the schedule

containing it must take time at least q; therefore, the schedule in phase $f - i$ is of length at least q. We have at most m jobs; therefore, we can schedule one job per machine, and the length of the schedule in phase $f - i - \ell \log(2R)$ is at most $q/((2R)^{\ell} s_m)$. Since $s_m = 1/R$, the length of the schedule in phase $f - i - \ell \log(2R)$ is at most $q/2^{\ell}$. □

Combining Lemmas 3.6 and 3.8 we obtain the following theorem.

THEOREM 3.9. *There is a (polynomial time) on-line algorithm for scheduling (preemptive/nonpreemptive) uniformly related machines that has competitive ratio* $O(\min(\log m, \log R))$.

How many preemptions/restarts does this general algorithm perform? In each iteration it is possible that no job finished and therefore there are n preemptions/restarts at the end of the iteration. If $p_{\min}$ is the minimum job size and $r = p_{\max}/p_{\min}$, the on-line algorithm does $O(\log(r\rho n))$ iterations. This is because the Δ established in step 1 can be no smaller than $p_{\min}$; we then set $q = \Delta/\rho n$ and successively double it until we reach $p_{\max}$. Therefore, the algorithms for unrelated machines do $O(n \log(n\rho r))$ preemptions/restarts, and those for uniformly related machines do $O(m \log(mr))$.

3.2. A different algorithm for nonpreemptive uniformly related machines. In this section we give a second $O(\log R)$-competitive algorithm for the nonpreemptive scheduling of uniformly related machines. This algorithm uses a new and simple off-line 2-approximation algorithm for uniformly related machines.

3.2.1. A simple (off-line) 2-relaxed decision procedure for uniformly related machines. First we give a new (off-line) 2-relaxed decision procedure for uniformly related machines that will be the basis of our on-line algorithm. The notion of a *ρ-relaxed decision procedure* was used by Hochbaum and Shmoys [23]: given a deadline d, such a procedure either produces a schedule of length ρd or verifies that there exists no schedule of length d. By using binary search, a ρ-relaxed decision procedure can be converted into a ρ-approximation algorithm.

The 2-relaxed decision procedure is as follows. Each machine has an associated queue. Each job is placed into the queue of the slowest machine m_k such that $p_j \le s_k d$, that is, the slowest machine that can complete the job within the given deadline. If, for some job, there is no such machine, it is clear that there does not exist a schedule of length d. To construct a schedule, whenever a machine is idle, it starts processing a new, unprocessed job from its queue. If a machine's queue is empty, it takes a job to process from the queue of the fastest machine that is slower than it and has a nonempty queue. If all such queues are empty, then the machine remains idle. If the schedule constructed has $C_{\max} > 2d$, output **no**. Otherwise we have produced a schedule of length at most $2d$.

To prove that this is a 2-relaxed decision procedure, we must prove that when the procedure outputs **no**, there is no schedule of length d. Consider a job j that was not finished by time $2d$. Since jobs are only processed by machines on which they take less than d units of time, this job must have started after time d; thus it was on the queue of some machine m_k until time d. This implies that, until time d, machines $m_1, \ldots, m_k$ were all busy processing jobs that could not possibly have been completed on machines $m_{k+1}, \ldots, m_m$ by time d. Therefore, in a schedule of length d it is impossible to process all of these jobs and job j, and so there exists no schedule of length d.

3.2.2. The on-line algorithm for nonpreemptive uniformly related machines. In this section we show how to convert this off-line relaxed decision procedure to an on-line algorithm.

THEOREM 3.10. *Let $\mathcal{I}$ be an instance of the scheduling problem for nonpreemptive uniformly related machines. Then there is an on-line scheduling algorithm which produces a schedule no longer than* $8(\log(2R))C^*_{\max}(\mathcal{I})$.

Proof. We round machine speeds down to the nearest power of two: when a machine finishes processing a job, it pretends to keep processing it long enough so that it seems to have been processed at the lesser speed. When we interpret the schedule for this rounded problem instance as a schedule for the actual problem instance, the competitive ratio can be increased by at most a factor of two. Since the rounded s_i are all powers of two and each rounded s_i is within a factor of at most $2R$ of the rounded s_1, it immediately follows that there are at most $\log(2R)$ different machine speeds. Let $M_1 = \{m_i | s_i = s_1\}$, $M_2 = \{m_i | s_i = s_1/2\}, \ldots, M_{\log(2R)} = \{m_i | s_i = s_1/2^{\log(2R)}\}$.

Our strategy will be first to convert the off-line 2-relaxed decision procedure into an on-line $2\log(2R)$-relaxed decision procedure, and then from that procedure build an on-line algorithm. The off-line decision procedure does not immediately lend itself to an on-line procedure, since the criterion it uses to assign jobs to machine queues utilizes knowledge of the job sizes. To convert this to an on-line decision procedure we will repeatedly run the off-line relaxed decision procedure to either schedule a job or else update the estimate of its size. Note that given the rounded machine speeds, instead of queueing jobs on machines $m_1, \ldots, m_m$, we can instead queue jobs on sets of machines $M_1, \ldots, M_{\log(2R)}$.

A formal description of an on-line $2\log(2R)$-relaxed decision procedure is as follows. The procedure either outputs **no** if there is no schedule of length d or it produces a schedule of length $2d\log(2R)$. Note that even if it answers **no**, the procedure may have completely processed some of the jobs in that time.

Input. A set of jobs and a deadline d.

Step 0. Put all jobs into the $M_{\log(2R)}$ queue.

Step 1. Run the off-line 2-relaxed decision procedure with the modification that no jobs are started after time d (that is, when a machine is idle it takes a job to process off of its queue or, when its queue is empty, off of the first slower machine that has a nonempty queue; etc.).

Step 2. 1. **If** all jobs finish processing by time $2d$ **stop**.

2. **If** any machine in M_1 is still processing a job at time $2d$ then there is no schedule of length d. Output **no**; **return**.

3. **If** any set of machines M_k has a job j in its queue at time d then there is no schedule of length d. Output **no**; **return**.

4. **If** there are jobs that are being processed at time $2d$, on machines in M_i, $i > 1$, cancel these jobs and put them on the queue of M_{i-1}. Go to Step 1.

To prove that this is an on-line $2\log(2R)$-relaxed decision procedure, note first that the length of the schedule or partial schedule produced by this procedure is no longer than $2d\log(2R)$, since the off-line relaxed decision procedure produces a schedule of length at most $2d$ and is run at most $\log(2R)$ times. Furthermore, despite the fact that the p_j are unknown, the on-line relaxed decision procedure maintains the invariant that a job is only on the queue of M_k if it could not complete in time d on any machine in $M_{k+1}, \ldots, M_{\log(2R)}$. This is certainly true initially, since all the jobs are put in the queue of $M_{\log(2R)}$. Furthermore, since the procedure does not start new jobs after time d, any job that is still being processed at time $2d$ on some machine in M_i must take more than time d to process on any machine in M_i. Therefore, any such job does not belong on the queue of M_i or any slower set of machines and is placed on the queue of M_{i-1}.

Now we will show that if the procedure outputs **no** then there is no schedule of length d. If condition 2 is true then a machine in M_1 ran a job for more than d units of time; therefore, this job could not have been processed in d units of time on any of the machines, since no other machine runs at a faster speed. If condition 3 is true, then up until time d, all machines in the

sets $M_1, \ldots M_k$ must have been busy processing jobs that could not have been processed in d units of time on machines in $M_{k+1}, \ldots, M_{\log(2R)}$. Therefore, machines in $M_1, \ldots, M_k$ could not have processed all of these jobs and job j as well by time d.

We have given an on-line $2\log(2R)$-relaxed decision procedure; we now show how to use it to develop an on-line $O(\log R)$-competitive algorithm.

The on-line algorithm initially establishes a lower bound Δ on $C^*_{\max}$ by running an arbitrarily chosen job on the fastest processor. Let Δ be the time taken to complete this job; certainly $\Delta \leq C^*_{\max}$. Next, the on-line algorithm calls the procedure on the set of all jobs with $d = \Delta$. If the procedure returns **no**, then we will call it again with $d = 2\Delta$ and the set of jobs that were not completely processed in the first iteration. In general, if the ith iteration fails to produce a schedule, then we will call the procedure again for the $(i+1)$st time with $d = 2^i\Delta$ and all jobs that have not yet been completely processed. Observe that if the ith iteration fails to produce a schedule when called with $d = 2^{i-1}\Delta$, then it proves that $2^{i-1}\Delta < C^*_{\max}$. Suppose that we finally finish processing all jobs in iteration f. Then the total length of the schedule produced is

$$\Delta + (1 + 2 + \cdots + 2^f)(2\Delta \log(2R)) \leq 2^{f+2}\Delta \log(2R).$$

Since the procedure failed to produce a schedule in iteration $f-1$, we know that $2^{f-1}\Delta < C^*_{\max}$. Therefore, the total length of the schedule produced is no greater than $8(\log(2R))C^*_{\max}$. □

This algorithm can also be modified when $R > m$ to obtain an $O(\min(\log R, \log m))$-competitive algorithm; we simply ignore those machines with $s_i < s_1/m$ and note that this increases the optimum makespan for any instance by at most a factor of 2. To bound the number of restarts of the above on-line algorithm, observe that in any iteration of the on-line relaxed decision procedure, $O(m)$ jobs will be restarted $O(\log R)$ times. The on-line relaxed decision procedure is run at most $O(\log(C^*_{\max}/p_{\min}))$ times, since the initial candidate deadline is at least $p_{\min}$ and we successively double the deadline until we reach a feasible deadline, which $C^*_{\max}$ certainly is. Therefore, this algorithm performs $O(m \log R \log(C^*_{\max}/p_{\min}))$ restarts.

4. Lower bounds.

4.1. Introduction. As with other on-line algorithms, on-line scheduling can be viewed as a game against an adversary who is allowed to determine the information that is revealed incrementally to the algorithm. Therefore, our lower bound arguments will be phrased in terms of a strategy for an adversary, who attempts to reveal information in such a way so as to force the competitive ratio to be as large as possible.

In many cases, we will give the scheduling algorithm more power than our model allows and prove that any such algorithm has competitive ratio at least ρ, thereby implying that any algorithm in our model must also have at least that competitive ratio. For example, we will sometimes assume that the algorithm is given the multiset of processing times but not the correspondence between the jobs and their processing times.

We will prove lower bounds in both preemptive and nonpreemptive scheduling models; in the latter case, we shall always assume that the algorithm is allowed to restart jobs. Our lower bounds are purely information theoretic; they rely on no complexity assumptions.

In proving lower bounds, it is important to specify fully the algorithmic primitives that may be used by the algorithm to acquire information about the input. In our models, the algorithm is allowed to ask the adversary whether any job will complete by a given time t if the present set of jobs remains assigned to their given machines; the adversary must either answer "no" or give the time $t' \leq t$ that the next job completes, and give all jobs that complete at time t'. In the two cases, respectively, the schedule until time t and t' is then fixed.

4.2. Release dates. We begin our lower bounds with the proof of Theorem 2.3. This theorem concerns the model in which the release dates are unknown, but once a job is released, its processing requirement is known. In this model, we can view the interaction between the algorithm and the adversary in the following way: the algorithm specifies the schedule until a given time t and then asks the adversary if any jobs are released by time t; the adversary must either answer "no" or give the time $t' \leq t$ that the next job is released, and give all jobs that are released at time t'. In the two cases, respectively, the schedule until time t and t' is then fixed.

We shall assume that the algorithm is given the information that the processing requirements and release dates are all integral: any lower bound proven with this additional information must also hold for the general case. As a consequence of this assumption, it is natural to believe that any reasonable algorithm will construct a schedule in which each job has integral starting times. Our proofs, however, will rely on significantly weaker assumptions which can be made without loss of generality. We can assume that the algorithm initially commits itself to a schedule until time 1: since no jobs are released until then, the adversary provides no information until that time; this implies that any algorithm that starts a job within the open interval (0, 1) is dominated by the same algorithm that starts the job at time 0. In particular, we can assume that there are no restarts in (0, 1). Furthermore, we will use the notion of the *integral remaining work* at some time t in a given schedule: for each job, compute the floor of its processing requirement that remains unprocessed and then compute the sum of these values over all jobs. Observe that if the integral remaining work at time t is W, then the schedule cannot complete earlier than time $t + \lceil W/m \rceil$.

THEOREM 2.3 (restated). *Let $\mathcal{A}$ be a deterministic on-line algorithm for nonpreemptive scheduling of identical machines with unknown release dates but known processing requirements. Then the competitive ratio of algorithm $\mathcal{A}$ is at least $\frac{10}{9}$, even if restarts are allowed.*

Proof. Consider the instance that consists of two identical machines and three jobs released at time 0: jobs A and B of size 3 and job C of size 2. We consider several cases.

(i) The algorithm initially schedules job A on machine 1 and job B on machine 2 at time 0 and decides to leave them both uninterrupted until at least time 2 (except if another job is released earlier). In this case, the adversary releases job D of size 4 at time 2. The optimal schedule for this instance processes jobs A and B on one machine and job C followed by job D on the other; it has length 6. If the algorithm interrupts job A or job B at time 2, then the integral remaining work at time 2 is at least 10, and so the schedule must be of length at least 7. Clearly, interrupting A or B in (2, 3) has no advantage over interrupting them at time 2. Any schedule that does not interrupt A or B and then processes job D must also be of length at least 7. Hence, in this case, the performance guarantee is at least $\frac{7}{6}$.

(ii) The algorithm initially schedules job A on machine 1 and job B on machine 2 at time 0 and decides to interrupt at least one of them before time 2. In this case, the adversary releases job D of size 2 at time 2. This implies that at time 2, the integral remaining work is at least 7: the total work of this instance is 10 processing units, and at most 3 units of (completed) processing occurred before time 2, since there are no restarts in (0, 1). Hence the schedule must be of length at least 6. However, the optimal schedule is of length 5: one machine processes jobs A and C; the other processes job B followed by D. Hence the performance ratio is at least $\frac{6}{5}$.

(iii) The algorithm only schedules one job to begin at time 0. In this case the adversary releases a job of size 2 at time 1. The integral remaining work at time 1 is at least 9, and hence the length of the schedule is at least 6. The resulting performance ratio is at least $\frac{6}{5}$.

(iv) The algorithm schedules jobs A and C at time 0. The adversary then releases a job D of size 7 at time 1. If the algorithm decides to let job C complete, then the minimum length

of the produced schedule is 9, whereas the optimal schedule would be of length 8; hence the performance bound is at least $\frac{9}{8}$. If the algorithm decides to interrupt job A or job C before time 2, then the adversary releases job E of size 3 at time 3. The optimal schedule for this instance is of length 9: one machine can process all jobs of length 3, and the other machine processes jobs C and D. However, the integral remaining work at time 3 is at least 13, and hence the length of the schedule is at least 10. Therefore the performance bound in this case, and in general, is at least $\frac{10}{9}$. □

4.3. Identical machines.

THEOREM 4.1. *The competitive ratio of any deterministic on-line algorithm for scheduling identical machines, with no preemption allowed, is at least* $(2 - \frac{1}{m})$.

Proof. For any m, let $n = m(m-1) + 1$. Each of the first $m(m-1)$ jobs is of size 1, while the last job is of size m; that is, $p_1 = \cdots = p_{n-1} = 1$, $p_n = m$. This instance is due to Graham [19]. The optimal schedule is of length m and consists of scheduling the last job on a machine by itself and scheduling m of the single unit jobs on each of the remaining $m-1$ machines. The length of a schedule for this instance is determined by the starting time of the job of size m; therefore, the adversary wishes to make it start as late as possible. Each of the first $n-1$ jobs that the algorithm allows to run for at least one unit of time will be fixed by the adversary to be jobs of size 1. Given this strategy for the adversary, it is not difficult to see that by time i, $i = 1, \ldots, m-1$, at most im jobs are either completely processed or currently being processed. Hence, at time $m-1$ there must be at least one job that has not been completely processed and is not currently being processed. The adversary sets this job to be of size m. Therefore, the schedule must be of length at least $2m-1$, which is $(2 - \frac{1}{m})$ times as long as the optimal schedule. □

In contrast to the nonpreemptive model, an optimal preemptive schedule can be found off-line in polynomial time [33]. Interestingly enough, an argument similar to the previous proof shows that the on-line worst-case characterization of both models is the same.

THEOREM 4.2. *Any deterministic on-line algorithm for scheduling identical machines with preemption allowed has competitive ratio at least* $(2 - \frac{1}{m})$.

Proof. Consider the behavior of the algorithm on an instance with $n = m + 1$ jobs. Suppose that the algorithm first asks the adversary if there is a job that completes by time t'. If the algorithm's proposed schedule contains a preemption, let t be the time of the first preemption; otherwise let $t = t'$. The adversary then sets the processing time for each job as follows: let n be the number of a job which has not been started by time t; let $p_1 = \cdots = p_{n-1} = t$ and $p_n = tm/(m-1)$. The algorithm clearly cannot complete all of the jobs earlier than time $t + tm/(m-1)$. The length of the optimal preemptive schedule is known to be $\max(p_{\max}, \sum p_j/m)$ [33]. In this case $\max(p_{\max}, \sum p_j/m) = tm/(m-1)$. Therefore, the algorithm has competitive ratio $[t + tm/(m-1)]/[tm/(m-1)] = (2 - \frac{1}{m})$. □

The essence of these deterministic lower bounds is that there is one large job whose starting time determines the length of the schedule, and the adversary can force the algorithm to start that job late in the schedule. One approach to defeat such an adversary is to allow the algorithm to be randomized. In proving lower bounds for randomized algorithms, one must be careful to delimit the adversary's access to the random bits used by the algorithm. In our setting, it is most natural to consider an *oblivious adversary*, who knows only the algorithm but not the coin tosses [6, 36]. The oblivious adversary models the situation where a randomized algorithm receives a problem instance and must produce a solution; the random choices it makes have no effect on the input it sees.

A stronger model is an *adaptive adversary*, who knows in advance the scheduling algorithm and may set attributes of a job at time t based on the algorithm's actions up to time t

[6]. The adaptive adversary models the situation where there is some feedback between the choices the algorithm makes and future input it sees. A good example of this latter situation is paging: depending on what random choices a paging algorithm makes, it may or may not itself cause page faults in addition to those caused by other elements of the operating system. However, it is clear that for our models, a randomized algorithm running against an adaptive adversary is no more powerful than a deterministic algorithm.

A priori, it would not be surprising if a randomized algorithm $\mathcal{A}$ working against an oblivious adversary might, with significant probability, select and schedule the large jobs earlier, thus improving its performance. A randomized algorithm can clearly gain *something* over a deterministic algorithm: consider, for example, the algorithm that randomly chooses an ordering of the jobs and then list schedules according to that ordering. Since each list schedule is at most $(2 - \frac{1}{m})$ times optimal in length, and at least one of the list schedules will be the optimal schedule, this randomized algorithm has expected performance strictly less than $(2 - \frac{1}{m})$. We will prove, however, that randomness is ultimately of little help to a nonpreemptive on-line scheduling algorithm for identical machines.

THEOREM 4.3. *Any randomized on-line algorithm for nonpreemptive scheduling of identical machines, working against an oblivious adversary, has competitive ratio at least* $(2 - O(\frac{1}{\sqrt{m}}))C^*_{\max}$.

We will actually prove a slightly stronger fact: this theorem is true even if the on-line algorithm knows in advance the sizes of the jobs, but does not know which size is associated with each job. The instance $\mathcal{I}$ that we consider consists of $m(m-k)$ jobs of size 1 and k jobs of size m, where k is a parameter less than m that we will later choose to optimize the lower bound derived; in fact, we will set k to be roughly $\sqrt{m}$. The optimal schedule is clearly of length m. The length of any schedule for this instance is determined by the starting time of the last job of size m. The goal of the algorithm is therefore to find all large jobs and begin processing them. The algorithm cannot know whether a job is of size 1 or of size m until it has processed it for one unit of time.

Therefore, the important information about the execution of the algorithm can be captured by a sequence S of points in time at which each job's first unit of processing time is completed. We will assume that when an algorithm discovers a big job it lets it run to completion without further interruption. The possibility that a shorter schedule could arise from postponing further processing of the large jobs does not affect our analysis, since we will give the algorithm the extra power of processing the first unit of m jobs at every unit of time even when some large jobs are concurrently being processed. The goal of the algorithm is to have the processing of the first unit of the last job of size m be as early as possible in the sequence S.

A randomized algorithm for this problem can then be described as choosing with a certain probability distribution one of the remaining jobs to be the next one to be run for one unit of time. The probability can depend on what has been seen before: this yields information about how many jobs of size 1 and size m remain, but gives no information to distinguish among the unprocessed jobs.

We will now argue that the adversary can always force the algorithm to do as poorly as it would have done had it always made its choices according to the uniform distribution. The following lemma, while different in detail, is in the spirit of the result of Yao [7], which uses the minimax theorem of Von Neumann to prove a worst-case lower bound for randomized algorithms based on an average-case lower bound for deterministic algorithms on probabilistic inputs.

LEMMA 4.4. *The competitive ratio of a randomized algorithm $\mathcal{A}$ can be no less than that of the algorithm U that always picks the next job to process uniformly from among the remaining jobs.*

Proof. We note that the adversary's strategy can be described as choosing some permutation of the jobs. If the adversary chooses the permutation randomly and uniformly, then the probability of the algorithm $\mathcal{A}$ selecting any particular job is uniform over all remaining jobs, no matter what probability distribution $\mathcal{A}$ uses. Let $\mathcal{E}$ be the expected performance of algorithm $\mathcal{A}$ against this adversary, where the expectation is taken over the random choices of both $\mathcal{A}$ and the adversary. Note that the adversary can always choose some permutation of jobs such that the expected performance of $\mathcal{A}$, taken over just the choices of $\mathcal{A}$, is no better than $\mathcal{E}$. Since the expected performance of the algorithm U that chooses uniformly is $\mathcal{E}$ no matter which permutation is used, algorithm $\mathcal{A}$ can have competitive ratio no better than algorithm U. □

We complete the proof of Theorem 4.3 by showing that scheduling by choosing the next job uniformly can do quite poorly.

LEMMA 4.5. *If instance $\mathcal{I}$ is scheduled on identical machines by choosing the next job to be processed for one time unit uniformly at random, the expected length of the schedule produced is at least* $(2 - O(\frac{1}{\sqrt{m}}))C^*_{\max}$.

Proof. The expected length of the schedule is then $m + E_s$, where E_s is the expected start time of the last job of size m in the schedule. To bound E_s, we can think of the problem as a "shell game," where there are $n = m(m-k) + k$ shells, under k of which there are peas. We show that if one searches for the peas by choosing among the remaining shells randomly and uniformly, the expected place of the last pea to be found is $\frac{k}{k+1}(n+1)$. The searcher can be described as using a uniformly chosen random permutation of the n positions; therefore, the expected time of finding the last pea is the same as the expected position E_k of the last pea if the peas were uniformly distributed among the shells according to a random permutation.

To compute E_k, consider, for convenience, the symmetric problem of the expected position of the *first* pea, E_1 (note that we number the first position as 1, not 0):

$$E_1 = \sum_{l=1}^{n-k+1} \text{Prob[Position of first pea} \geq l] = \sum_{l=1}^{n-k+1} \frac{\binom{n-l+1}{k}}{\binom{n}{k}} = \frac{\binom{n+1}{k+1}}{\binom{n}{k}} = \frac{n+1}{k+1}.$$

By symmetry the expected position of the last pea is $\frac{k}{k+1}(n+1)$.

Therefore, we expect the kth job of size m to be the $\frac{k}{k+1}[m(m-k)+k]$th chosen overall. This will happen no earlier than time $\lfloor \frac{k}{k+1}(m-k) \rfloor$, since at most m jobs are completed during every unit of time; as a result $E_s \geq \lfloor \frac{k}{k+1}(m-k) \rfloor = F(k)$. We drop the floor, which has no impact on the asymptotic quality of our bounds. To derive the strongest lower bound possible we maximize $F(k)$ by using elementary calculus. The maximum is achieved by setting $k = \sqrt{m+1} - 1$; plugging into $F(k)$ we see that

$$\begin{aligned} F(k) &= \frac{\sqrt{m+1}-1}{\sqrt{m+1}}\left(m - \sqrt{m+1} + 1\right) \\ &> \left(1 - \frac{1}{\sqrt{m+1}}\right)\left(m - \sqrt{m+1}\right) \\ &> m - \frac{m}{\sqrt{m+1}} - \sqrt{m+1} \\ &> m - 3\sqrt{m}. \end{aligned}$$

Thus $m + E_s > 2m - 3\sqrt{m} = (2 - \frac{3}{\sqrt{m}})C^*_{\max}$, which implies the stated result. □

4.4. Uniformly related machines. In the case of uniformly related machines the situation becomes significantly more difficult for the scheduler. We will show that an adversary can force any deterministic scheduler to construct a schedule of length $\Omega(\log m)$ times the length of the optimal schedule, whether or not the scheduler is allowed to preempt jobs.

THEOREM 4.6. *The competitive ratio of any deterministic on-line scheduling algorithm for uniformly related machines, whether or not preemption is allowed, is* $\Omega(\log m)$.

To prove this theorem we use a family of instances $\mathcal{I}_k$ that was introduced by Cho and Sahni [9] to show that a certain off-line approximation algorithm had a poor performance guarantee. Let $k = (\log_2(3m-1)+1)/2$. We restrict ourselves to values of m such that k is integral. The instance $\mathcal{I}_k$ has m jobs and m machines; there are k sets of machines G_i and k sets of jobs T_i, $i = 1, \ldots, k$. Each machine in G_i has a speed of 2^i and each job in T_i has size 2^i. Finally, $|G_i| = |T_i| = 2^{2k-2i-1}$, $i = 1, \ldots, k-1$, and $|G_k| = |T_k| = 1$. It is easy to see that $C^*_{\max} = 1$, since each job of size 2^i can be scheduled by itself on a machine of speed 2^i, $i = 1, \ldots, k$.

Once again, we shall assume that the algorithm is given the multiset of job sizes for the input, but is not given the correspondence between each job and its size. The proof of the theorem is based on a rather natural strategy for the adversary, which we call the *delayed commitment strategy*. For any time t, let $J(t)$ denote the set of jobs that have not yet been completed and let $L(t)$ denote the multiset of the sizes of the jobs in $J(t)$. We shall say that the set $J(t)$ is *uncommitted* in a particular schedule at time t if, for any bijection $\rho : J(t) \to L(t)$, there is an extension of the schedule so that each job $j \in J(t)$ is processed for $\rho(j)$ time units. Equivalently, the schedule up to time t implies a lower bound on the size of each job $j \in J(t)$, which we shall denote $p_j(t)$; the set $J(t)$ is uncommitted if

$$\max_{j \in J(t)} p_j(t) \leq \min_{p \in L(t)} p. \tag{1}$$

If this is a strict inequality, we shall say that $J(t)$ is *strictly uncommitted* at time t. The aim of the delayed commitment strategy is to ensure that at any time t, the set $J(t)$ is uncommitted in the schedule produced by the algorithm. The strategy works as follows: whenever $J(t)$ is strictly uncommitted, the algorithm continues to produce the schedule; whenever this first fails to be true, and $J(t)$ is only uncommitted, there exists a job j such that $p_j(t) = \min_{p \in L(t)} p$, and j is then assigned processing time $\min_{p \in L(t)} p$. This causes j to be deleted from $J(t)$ and p_j to be deleted from $L(t)$. If inequality (1) is still tight, then this is repeated until the remaining set $J(t)$ is once again strictly uncommitted. When inequality (1) is tight because of more than one job, we shall, for concreteness, assume that the tight job of minimum index is selected.

LEMMA 4.7. *Consider the execution of any scheduling algorithm on the instance* $\mathcal{I}_k$ *when the processing times are assigned by the delayed commitment strategy. Let* X_i *denote the time when the last job in* T_i *finishes,* $i = 1, \ldots, k$, *and let* $X_0 = 0$. *Then* $X_0 \leq X_1 \leq X_2 \leq \cdots \leq X_k$ *and, furthermore,* $X_{i+1} - X_i \geq \frac{1}{4}$, $i = 0, \ldots, k-1$.

Proof. We first show that $X_i \leq X_{i+1}$ for each $i = 0, \ldots, k-1$. In fact, we will show something stronger: for each $i = 0, \ldots, k-1$ at time X_i, each job $j \in T_{i+1}$ has at least 2^i units of processing remaining. The case $i = 0$ is trivial, and so we consider each $i = 1, \ldots, k-1$. Let j' be the last job in T_i to be completed; job j' completes at time X_i. Consider any job j in T_{i+1} and suppose that more than 2^i units of its processing requirement have been completed by the schedule at time X_i. This implies that at some earlier point in time t, exactly 2^i units were completed. However, since j' was not yet assigned its processing time at time t, it follows that $2^i \in L(t)$, and so j would be assigned processing time 2^i instead. This contradiction implies that at time X_i, each job $j \in T_{i+1}$ has had at most 2^i units of its processing requirement already completed; hence, at least 2^i units of processing remain.

We have just argued that each of the $2^{2k-2i-3}$ jobs in T_{i+1} has at least 2^i units of processing remaining at time X_i, $i = 0, \ldots, k-1$. We will use this fact to derive a lower bound on the time that must subsequently elapse before all jobs in T_{i+1} can complete. An easy lower bound is given by the ratio of the total remaining processing requirement to the total machine speed that can be used. Since there are more than $|T_{i+1}|$ machines, we can assume that only the $|T_{i+1}|$ fastest machines will now be used to process the jobs in T_{i+1}. It is easy to see that there are more than $|T_{i+1}|$ machines in the sets $G_{i+1}, G_{i+2}, \ldots G_k$. Hence we can lower bound the time that elapses until all jobs in T_{i+1} are completed by

$$\frac{2^i|T_{i+1}|}{\sum_{l=i+1}^{k} 2^l|G_l|} = \frac{2^{2k-i-3}}{\sum_{l=i+1}^{k-1} 2^l \cdot 2^{2k-2l-1} + 2^k}$$

$$= \frac{2^{2k-i-3}}{2^k \sum_{r=0}^{k-i-2} 2^r + 2^k}$$

$$= \frac{2^{2k-i-3}}{2^k(2^{k-i-1} - 1) + 2^k}$$

$$= \frac{2^{2k-i-3}}{2^{2k-i-1}}$$

$$= 1/4. \qquad \square$$

This lemma implies that the schedule completes at time $X_k \geq \frac{k}{4} = \Omega(\log m)$. Hence we have an $\Omega(\log m)$ lower bound on the competitive ratio.

THEOREM 4.8. *The competitive ratio for any deterministic on-line algorithm for scheduling uniformly related machines with $R = s_1/s_m < m$ is $\Omega(\log R)$, whether or not preemption is allowed.*

Proof. We modify our previous proof slightly by providing a nearly identical family of instances that satisfies $s_1/s_m \leq R$. Let $r = \lfloor \log_2 R \rfloor$; the new family will have $s_1/s_m = 2^r$. In the instance $\mathcal{I}_k$, the fastest machine has speed 2^k. We modify $\mathcal{I}_k$ to meet the speed restriction by increasing the speed of all machines of speed less than 2^{k-r} to be exactly 2^{k-r}. This yields our new family of instances. The optimal schedule length is still 1. We can also prove an analogue of Lemma 4.7 for the delayed commitment strategy. As before, if we let X_i denote the time that the last job in T_i is completed, this strategy ensures that $X_1 \leq X_2 \leq \cdots \leq X_k$. Furthermore, the identical proof still yields that $X_i - X_{i-1} \geq \frac{1}{4}$ for each $i = k-r, \ldots, k$. Intuitively, all of the "small" jobs may finish quite quickly, but those in $T_{k-r}, \ldots, T_k$ will each take at least $\frac{1}{4}$ unit of time to complete. $\square$

5. Conclusions and open problems. The most obvious open problem raised by our work is to close the gap between the upper bound of $O(\log n)$ and the lower bound of $\Omega(\log m)$ for unrelated machines. To do this, we would need only a "preprocessing algorithm" that reduced the number of jobs to a number polynomial in m. For uniformly related machines, we showed that list scheduling of the first $n - m$ jobs accomplishes this goal. It is not clear, however, that a similar naive approach will be of use for unrelated machines.

It has been observed in the context of paging and list maintenance that the conclusions drawn from average-case analysis of on-line algorithms do not always correspond to the conclusions of experimental studies and practical experience [39]. If this proves to be the case for parallel machine scheduling, then the design and analysis of a model that is less pessimistic

than the worst-case competitive ratio but has more structure than expected performance on randomly selected instances might be a valuable endeavor.

Acknowledgments. We are grateful to Howard Karloff for suggesting on-line scheduling as a fruitful area of research. We would like to thank Serge Plotkin for allowing us to include his observation that we could obtain an $O(\log R)$-competitive algorithm directly from our general technique. We also thank Nabil Kahale for a useful discussion on probability and Cliff Stein, Lisa Hellerstein, and the two anonymous referees for comments on earlier versions of this paper.

REFERENCES

[1] J. ASPNES, Y. AZAR, A. FIAT, S. PLOTKIN, AND O. WAARTS, *On-line load balancing with applications to machine scheduling and virtual circuit routing*, in Proc. 25th Annual ACM Symposium on Theory of Computing, 1993, pp. 623–631.

[2] B. AWERBUCH, S. KUTTEN, AND D. PELEG, *Competitive distributed job scheduling*, in Proc. 24th Annual ACM Symposium on Theory of Computing, 1992, pp. 571–581.

[3] Y. AZAR, A. BRODER, AND A. KARLIN, *On-line load balancing*, in Proc. 33rd Annual IEEE Symposium on Foundations of Computer Science, 1992, pp. 218–225.

[4] Y. AZAR, J. NAOR, AND R. ROM, *The competitiveness of on-line assignments*, in Proc. 3rd Annual ACM-SIAM Symposium on Discrete Algorithms, 1992, pp. 203–210.

[5] Y. BARTAL, A. FIAT, H. KARLOFF, AND R. VOHRA, *New algorithms for an ancient scheduling problem*, in Proc. 24th Annual ACM Symposium on Theory of Computing, 1992, pp. 51–58.

[6] S. BEN-DAVID, A. BORODIN, R. M. KARP, G. TARDOS, AND A. WIGDERSON, *On the power of randomization in on-line algorithms*, in Proc. 22nd Annual ACM Symposium on Theory of Computing, 1990, pp. 379–386.

[7] A. C.-C. YAO, *Probabilistic computations: Toward a unified measure of complexity*, in Proc. 18th Annual IEEE Symposium on Foundations of Computer Science, 1990, pp. 222–227.

[8] B. CHANDRA, H. J. KARLOFF, AND S. VISHWANATHAN, 1991, private communication.

[9] Y. CHO AND S. SAHNI, *Bounds for list schedules on uniform processors*, SIAM J. Comput., 9 (1980), pp. 91–103.

[10] E. DAVIS AND J. M. JAFFE, *Algorithms for scheduling tasks on unrelated processors*, J. Assoc. Comput. Mach., 28 (1981), pp. 712–736.

[11] X. DENG AND E. KOUTSOUPIAS, *Competitive implementation of parallel programs*, in Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms, 1993, pp. 455–461.

[12] U. FAIGLE, W. KERN, AND G. TURÁN, *On the performance of on-line algorithms for partition problems*, Acta Inform., 9 (1989), pp. 107–119.

[13] A. FELDMANN, M.-Y. KAO, J. SGALL, AND S.-H. TENG, *Optimal online scheduling of parallel jobs with dependencies*, in Proc. 25th Annual ACM Symposium on Theory of Computing, 1993, pp. 642–651.

[14] A. FELDMANN, J. SGALL, AND S.-H. TENG, *Dynamic scheduling on parallel machines*, in Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science, 1991, pp. 111–120.

[15] A. FIAT, Y. RABANI, AND Y. RAVID, *Competitive k-server algorithms*, in Proc. 31st Annual IEEE Symposium on Foundations of Computer Science, 1990, pp. 454–463.

[16] G. GALAMBOS AND G. WOEGINGER, *An on-line scheduling heuristic with better worst case ratio than Graham's list scheduling*, SIAM J. Comput., 22 (1993), pp. 349–355.

[17] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, New York, 1979.

[18] T. GONZALEZ AND D. B. JOHNSON, *A new algorithm for preemptive scheduling of trees*, J. Assoc. Comput. Mach., 27 (1980), pp. 287–312.

[19] R. L. GRAHAM, *Bounds for certain multiprocessor anomalies*, Bell System Tech. J., 45 (1966), pp. 1563–1581.

[20] R. L. GRAHAM, E. L. LAWLER, J. K. LENSTRA, AND A. H. G. RINNOOY KAN, *Optimization and approximation in deterministic sequencing and scheduling: A survey*, Ann. Discrete Math., 5 (1979), pp. 287–326.

[21] D. GUSFIELD, *Bounds for naive multiple machine scheduling with release times and deadlines*, J. Algorithms, 5 (1984), pp. 1–6.

[22] L. A. HALL AND D. B. SHMOYS, *Approximation schemes for constrained scheduling problems*, in Proc. 30th Annual IEEE Symposium on Foundations of Computer Science, 1989, pp. 134–141.

[23] D. S. HOCHBAUM AND D. B. SHMOYS, *Using dual approximation algorithms for scheduling problems: Theoretical and practical results*, J. Assoc. Comput. Mach., 34 (1987), pp. 144–162.

[24] ———, *A polynomial approximation scheme for machine scheduling on uniform processors: Using the dual approximation approach*, SIAM J. Comput., 17 (1988), pp. 539–551.

[25] E. C. HORVATH, S. LAM, AND R. SETHI, *A level algorithm for preemptive scheduling*, J. Assoc. Comput. Mach., 24 (1977), pp. 32–43.

[26] S. IRANI, *Coloring inductive graphs on-line*, in Proc. 31st Annual IEEE Symposium on Foundations of Computer Science, 1990, pp. 470-479.

[27] J. M. JAFFE, *Efficient scheduling of tasks without full use of processor resources*, Theoret. Comput. Sci., 12 (1980), pp. 1-17.

[28] A. R. KARLIN, M. S. MANASSE, L. RUDOLPH, AND D. D. SLEATOR, *Competitive snoopy caching*, Algorithmica, 3 (1988), pp. 79-119.

[29] R. M. KARP, U. V. VAZIRANI, AND V. V. VAZIRANI, *An optimal algorithm for on-line bipartite matching*, in Proc. 22nd Annual ACM Symposium on Theory of Computing, 1990, pp. 352-358.

[30] E. L. LAWLER AND J. LABETOULLE, *On preeemptive scheduling of of unrelated parallel processors by linear programming*, J. Assoc. Comput. Mach., 25 (1978), pp. 612-619.

[31] J. K. LENSTRA, D. B. SHMOYS, AND É. TARDOS, *Approximation algorithms for scheduling unrelated parallel machines*, Math. Programming, 46 (1990), pp. 259-271.

[32] M. S. MANASSE, L. A. MCGEOCH, AND D. D. SLEATOR, *Competitive algorithms for server problems*, J. Algorithms, 11 (1990), pp. 208-230.

[33] R. MCNAUGHTON, *Scheduling with deadlines and loss functions*, Management Science, 6 (1959), pp. 1-12.

[34] R. MOTWANI, S. PHILLIPS, AND E. TORNG, *Non-clairvoyant scheduling*, in Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms, 1993, pp. 422-431.

[35] P. R. NARAYANAN AND R. CHANDRASEKARAN, *Optimal on-line algorithms for scheduling*, 1991, unpublished manuscript.

[36] P. RAGHAVAN AND M. SNIR, *Memory versus randomization in on-line algorithms*, IBM J. Res. Develop., 38 (1994), pp. 683-704.

[37] S. SAHNI AND Y. CHO, *Nearly on line scheduling of a uniform processor system with release times*, SIAM J. Comput., 8 (1979), pp. 275-285.

[38] D. B. SHMOYS, C. STEIN, AND J. WEIN, *Improved approximation algorithms for shop scheduling problems*, SIAM J. Comput., 23 (1994), pp. 617-632.

[39] D. D. SLEATOR AND R. E. TARJAN, *Amortized efficiency of list update and paging rules*, Comm. Assoc. Comput. Mach., 28 (1985), pp. 202-208.

[40] S. VISHWANATHAN, *Randomized online graph coloring*, J. Algorithms, 13 (1992), pp. 657-669.

SIAM J. COMPUT.
Vol. 24, No. 6, pp. 1332-1358, December 1995

012

COMPUTING MINIMAL SPANNING SUBGRAPHS IN LINEAR TIME*

XIAOFENG HAN†, PIERRE KELSEN‡, VIJAYA RAMACHANDRAN§, AND ROBERT TARJAN¶

Abstract. Let P be a property of undirected graphs. We consider the following problem: given a graph G that has property P, find a minimal spanning subgraph of G with property P. We describe general algorithms for this problem and prove their correctness under fairly weak assumptions about P. We establish that the worst-case running time of these algorithms is $\Theta(m + n \log n)$ for 2-edge-connectivity and biconnectivity where n and m denote the number of vertices and edges, respectively, in the input graph. By refining the basic algorithms we obtain the first linear time algorithms for computing a minimal 2-edge-connected spanning subgraph and for computing a minimal biconnected spanning subgraph.

We also devise general algorithms for computing a minimal spanning subgraph in directed graphs. These algorithms allow us to simplify an earlier algorithm of Gibbons, Karp, Ramachandran, Soroker, and Tarjan for computing a minimal strongly connected spanning subgraph. We also provide the first tight analysis of the latter algorithm, showing that its worst-case time complexity is $\Theta(m + n \log n)$.

Key words. minimal subgraphs, biconnectivity, two-edge-connectivity, strong connectivity, linear-time algorithm, worst-case behavior

AMS subject classifications. 05C85, 05C40, 68Q25, 68R10

1. Introduction.

Let P be a monotone graph property. In this paper we consider the following problem: given a graph G having property P, find a minimal spanning subgraph of G with property P, i.e., a spanning subgraph of G with property P in which the deletion of any edge destroys the property. We are interested in the sequential and parallel complexity of this problem.

The corresponding problem of finding a *minimum* spanning subgraph having a given property has been widely studied. We mention two results. Chung and Graham [3] proved that the problems of finding a minimum k-vertex-connected or k-edge-connected spanning subgraph are *NP*-hard for any fixed $k \geq 2$. For the more relaxed problem of finding sparse but not necessarily minimal k-edge-connected and k-vertex-connected spanning subgraphs, linear time algorithms are known [18]. Yannakakis ([24]; see also [15]) showed that the related problem of deleting a minimum set of edges so that the resulting graph has a given property is *NP*-hard for several graph properties (e.g., planar, outerplanar, transitive digraph).

There is a natural sequential algorithm for finding a minimal spanning subgraph with property P: examine the edges of G one at a time; remove an edge if the resulting graph has property P. This gives a polynomial time algorithm for the problem if the property P can be verified in polynomial time. However, for most nontrivial properties the running time of the algorithm is at least quadratic in the input size. Further, this algorithm seems hard to parallelize. Our goal is to obtain efficient sequential algorithms that can be parallelized effectively.

*Received by the editors December 21, 1991; accepted for publication (in revised form) July 15, 1994.

†Zyga Corporation, 28 West Oak Street, Baskingridge, New Jersey 07920. This research was done while this author was a graduate student at the Department of Computer Science, Princeton University, Princeton, New Jersey 08544.

‡Max Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany (kelsen@mpi-sb.mpg.de). This research was performed while this author was a graduate student at the Department of Computer Sciences, University of Texas, Austin, Texas 78712 and was supported in part by NSF grant CCR-89-10707.

§Department of Computer Sciences, University of Texas, Austin, Texas 78712 (vlr@cs.utexas.edu). The research of this author was supported in part by NSF grant CCR-89-10707.

¶Department of Computer Science, Princeton University, Princeton, New Jersey 08544 and NEC Research Institute, 4 Independence Way, Princeton, New Jersey 08540 (ret@cs.princeton.edu). The research of this author at Princeton University was partially supported by NSF grant CCR-8920505, ONR contract N00014-91-J-1483, and DIMACS, a National Science Foundation Science and Technology Center, grant NSF-STC88-09648.

The problem at hand may be phrased in the very general framework of *independence systems* described by Karp, Upfal, and Wigderson [12]: an *independence system* is a finite set together with a collection of subsets, called *independent sets*, with the property that any subset of an independent set is independent. Define a subset S of edges in G to be independent if the graph $G - S$ has property P. Finding a minimal spanning subgraph with property P amounts to finding a maximal independent set in the independence system that we have just defined. Efficient (deterministic) parallel algorithms for finding a maximal independent set in an independence system are known for the special case where the size of a minimal dependent set is 2 or 3 [16], [9], [5]. For the problems that are of interest to us, minimal dependent sets may have nonconstant size and hence a different approach is needed for obtaining fast parallel algorithms.

The minimal spanning subgraph problem has been studied earlier for the property of strong connectivity (*transitive compaction* problem [8]) and for 2-edge-connectivity and biconnectivity [13]. For these problems algorithms are given in [8], [13] that run in $O(m + n \log n)$ sequential time and can be implemented as NC algorithms; here n and m represent the number of vertices and edges in the input graph. Both papers have a similar high-level algorithm that is shown to terminate in $O(\log n)$ stages for the properties considered, and both papers leave open the question of whether this bound is tight.

In this paper we generalize the high-level algorithm of [8], [13] to a large class of graph properties. We show that for any graph property that implies 2-edge-connectivity the running time of these algorithms is within a logarithmic factor of the time required for minimally augmenting a spanning tree to achieve the given property. Because various computations on trees can be performed efficiently, both sequentially [22] and in parallel [17], [20], this algorithm provides a useful paradigm for the sequential and parallel determination of minimal spanning subgraphs with respect to connectivity properties.

We analyze the worst-case complexity of these algorithms. We show that the algorithms for 2-edge-connectivity and biconnectivity require $\Omega(\log n)$ iterations in the worst case; this implies that the worst-case time of these algorithms is $\Theta(m + n \log n)$, thus settling open questions posed in [13].

We describe refinements of the basic algorithms for 2-edge-connectivity and biconnectivity and obtain the first linear time algorithms for these properties. These algorithms still need a logarithmic number of iterations but by performing certain contractions and transformations on the current graph they reduce its size by a constant factor greater than 1 in a constant number of iterations. This result also reduces the work performed by the parallel algorithms for these problems by a logarithmic factor.

We also describe general algorithms for computing a minimal spanning subgraph in directed graphs with respect to any monotone property that implies strong connectivity. For the special case of strong connectivity, we are able to simplify the algorithm of [8] for computing a minimal strongly connected spanning subgraph. We also provide the first tight analysis of the latter algorithm, showing that its worst-case running time is $\Theta(m + n \log n)$. This answers a question posed in [8].

This paper is organized as follows. The next section defines the terms from graph theory used in this paper. In §3 we describe and analyze general algorithms for computing minimal spanning subgraphs in undirected graphs. In §4 we describe refinements that yield linear time algorithms for computing a minimal 2-edge-connected spanning subgraph and for computing a minimal biconnected spanning subgraph. In §5 we show that the basic algorithms have a worst-case time complexity of $\Theta(m + n \log n)$ if we do not incorporate those refinements. In §6 we develop and analyze general algorithms for computing minimal spanning subgraphs in directed graphs. In that section we also analyze an algorithm of [8] for computing a

minimal strongly connected spanning subgraph and we provide a simplified algorithm for the problem.

The sequential model of computation that we assume in this paper is the RAM [1] with a word length of $O(\log n)$ bits where n is the length of the input (or the number of vertices in the input graph).

2. Definitions. We introduce graph terminology similar to that of [2]. A *graph* is a triple $G = (V, E, \phi_G)$ where V is a set of *vertices*, and E, disjoint from V, is a set of *edges* (both V and E are assumed to be finite); the *incidence function* ϕ_G maps edges in E to unordered (ordered) pairs of vertices in V if the graph is undirected (directed). Note that this definition allows for the representation of multigraphs. We write $V(G)$ and $E(G)$ for the set of vertices and the set of edges, respectively, of G and use $n(G)$ and $m(G)$ to denote the number of vertices and edges, respectively, in G. We refer to a directed graph also as a *digraph*. We write (u, v) both for ordered and unordered pairs. If (u, v) is an edge in a directed graph, then u is called the *tail* of this edge and v is called the *head* of this edge. If $\phi(e) = (u, v)$, we say that edge e is *incident* on the vertices u and v and vertices u and v are the endpoints of e. The degree of a vertex v in G, denoted by $\deg_G(v)$, is the number of edges incident on vertex v. If the graph is directed, then the indegree (outdegree) of a vertex v is the number of edges $e \in E(G)$ such that $\phi_G(e) = (w, v)(\phi_G(e) = (v, w))$ for some vertex w.

If every edge in a graph joins two distinct vertices and no two edges join the same pair of vertices, then we say that the graph is *simple*. The *underlying graph* of a digraph G is obtained by omitting the directions of the edges in the digraph. We say that a digraph G is an *orientation* of an undirected graph G' if G' is the underlying graph of G.

The following definitions apply both to directed and undirected graphs. A *path* P in G is an alternating sequence $P = \langle v_0, e_1, v_1, \ldots, v_{k-1}, e_k, v_k \rangle$ of vertices and edges of G such that the following holds: (1) $\phi_G(e_i) = (v_{i-1}, v_i)$ for $0 < i \leq k$, (2) $v_0 \ldots v_{k-1}$ are distinct and $v_1 \ldots v_k$ are distinct; the integer k (number of edges) is the *length* of the path. The path P is a path *from* v_0 to v_k; v_0 and v_k are the *endpoints* of P and $v_1, \ldots, v_{k-1}$ are the *internal vertices* of P. A *subpath* of P is a path of the form $\langle v_i, e_{i+1}, \ldots, e_j, v_j \rangle$, where $0 \leq i \leq j \leq k$. The path P is a *chain* if it has length at least 2 and its internal vertices all have degree 2 in G. The path P is a *cycle* if $v_0 = v_k$.

If $G = (V, E, \phi_G)$ and $G' = (V', E', \phi_{G'})$ are two graphs such that $V' \subseteq V$, $E' \subseteq E$ and $\phi_{G'}$ is the restriction of ϕ_G to E', then G' is a *subgraph* of G; it is a *proper subgraph* if $G' \neq G$. If the graph G is understood, we may represent a subgraph H of G simply by the pair $(V(H), E(H))$. A *spanning subgraph* of G is a subgraph G' with $V(G') = V$. If $G' = (V', E')$ is a spanning subgraph of G and $E'' \subseteq E$, then $G' + E''$ denotes the subgraph $(V', E' \cup E'')$ of G and $G' - E''$ denotes the subgraph $(V', E' - E'')$ of G.

An undirected (directed) graph G is *connected* (strongly connected) if there exists a path from u to v for all $u, v \in V$. A subgraph of G is a *maximal* subgraph having a given property if it is not a proper subgraph of another subgraph of G with the same property. A *connected component* of an undirected graph G is a maximal connected subgraph of G. A *tree* in a graph G is a spanning connected subgraph of G without cycles. A *strong component* of a directed graph G is a maximal strongly connected subgraph of G.

All remaining definitions apply only to undirected graphs. A *cutedge* in G is an edge in G whose removal increases the number of connected components in G. The graph G is *2-edge-connected* if it is connected and has no cutedge or, equivalently, it is connected and every edge of G lies on a cycle. A *2-edge-connected component* of G is a maximal 2-edge-connected subgraph of G. A graph is *k-edge-connected* ($k > 0$) if $G - S$ is connected for every subset $S \subseteq E(G)$ of size less than k. A vertex v in G is a *cutpoint* if removing v together with all incident edges increases the number of connected components in G or, equivalently, there

exist distinct vertices u and w in G other than v such that any path from u to w contains v. A graph G is *biconnected* if it is connected, has at least three vertices, and does not contain a cutpoint. A *block of* G is a maximal subgraph of G with the property that it is connected and has no cutpoint. A graph G is *k-vertex-connected* if G has at least $k+1$ vertices and $G-S$ is connected for every $S \subseteq V(G)$ with $|S| < k$.

Let $G = (V, E, \phi_G)$ and let $V' \subseteq V$. The operation of *collapsing the vertices of V' in* G produces a graph G' defined as follows: $V(G') = (V - V') \cup \{z\}$ where z is a new vertex not in V; $E(G')$ is the subset of those edges in E that have at least one endpoint outside V'; $\phi_{G'}(e) = \phi_G(e)$ if no endpoint of e belongs to V' and $\phi_{G'}(e) = (z, v)$ if $\phi_G(e) = (u, v)$ where $u \in V'$ and $v \in V - V'$.

An *ear decomposition* [21] $D = [P_0, P_1, \ldots, P_{r-1}]$ of a graph G is a partition of $E(G)$ into an ordered collection of edge-disjoint paths $P_0, \ldots, P_{r-1}$ such that P_0 is a cycle and the two endpoints of P_i, for $i \geq 1$, are contained in lower-numbered ears, and none of the internal vertices of P_i are contained in lower-numbered ears. The paths in D are called *ears*. Ear decomposition D is an *open ear decomposition* if no P_i with $i > 0$ is a cycle. A *trivial ear* is an ear containing a single edge.

3. Finding a minimal spanning subgraph in undirected graphs. In this section we describe three closely related algorithms for finding a minimal spanning subgraph of a graph for various properties of undirected graphs. Algorithm 1 was first described in [14] while algorithms 2 and 3 are generalizations of algorithms for finding a minimal 2-edge-connected and a minimal biconnected spanning subgraph given in [10].

A *graph property* P is a Boolean-valued function on graphs. If $P(G)$ is true for some graph G, we say that G *has property* P or G is a *P-graph*. A *P-subgraph* of G is a subgraph of G that has property P. An edge e of a P-graph G is *P-redundant* in G if $G - e$ has property P, otherwise e is *P-essential* in G. We may not mention G or P if the graph or the property is clear from the context.

In this paper we concern ourselves with the problem of finding a minimal spanning P-subgraph of a P-graph G, i.e., a spanning P-subgraph of G in which every edge is P-essential. Throughout this paper we shall implicitly assume that property P is decidable, i.e., there is an algorithm that checks whether a given graph has property P. We restrict our attention to decidable properties that satisfy conditions C1 and C2 below:

C1. P is monotone, i.e., the addition of an edge to a P-graph results in a P-graph;

C2. any P-graph is connected.

As an immediate consequence of condition C1 we make the following basic observation.

OBSERVATION 1. *Let G be a P-graph and let H be a spanning P-subgraph of G. Any edge that is P-redundant in H is P-redundant in G.*

There is an obvious (sequential) algorithm for computing a minimal spanning P-subgraph of G: examine the edges of G one at a time; remove an edge if it is redundant in the current graph. By Observation 1 the resulting subgraph is a minimal spanning P-subgraph of G.

The following algorithm is a generalization of algorithms given in [13] and [8] (for finding a minimal 2-edge-connected, a minimal biconnected, and a minimal strongly connected spanning subgraph of a graph) to graph properties satisfying C1 and C2. This algorithm has been shown to outperform the obvious algorithm on undirected graphs for 2-edge-connectivity and biconnectivity, and we believe that this is true for a number of other properties of undirected graphs. Moreover, it is inherently easier to parallelize.

ALGORITHM 1. Computing a minimal spanning P-subgraph of G.

Input P-graph G.

Output Minimal spanning P-subgraph H of G.

(1) $H := G$;

(2) while H has P-redundant edges, do:
  (2.1) compute a spanning tree T_H in H with a maximum number of P-essential edges;
  (2.2) compute a minimal subset A of edges in H such that $T_H + A$ has property P;
  (2.3) $H := T_H + A$.

A spanning tree T_H of H containing a maximum number of essential edges (constructed in step 2.1) is called an *optimal tree in* H and the set A constructed in step 2.2 is called a *minimal augmentation for* T_H (in H).

THEOREM 1. *Algorithm* 1 *computes a minimal spanning* P*-subgraph of* G *for any property* P *satisfying* C1 *and* C2.

Proof. By induction on the number of iterations of the while-loop, one shows that H, as computed in step 2.3, is always a spanning P-subgraph of G. To prove termination, consider one execution of the while-loop. Since T_H is an optimal tree in H, it does not contain all redundant edges of H. Furthermore all edges of A are essential in $T_H + A$. Therefore, the number of redundant edges in H decreases by at least one at each iteration of the while-loop. Thus Algorithm 1 terminates. □

By the proof of Theorem 1 the number of iterations of Algorithm 1 is bounded by the number of edges in the input graph G. For several graph properties much sharper bounds hold. In this paper we are primarily interested in properties P that imply 2-edge-connectivity. The following theorem shows that for these properties Algorithm 1 terminates quickly. We use n to denote the number of vertices in G.

THEOREM 2. *If* P *satisfies* C1 *and* C2 *and also implies* 2*-edge-connectivity, then Algorithm* 1 *terminates after* $O(\log n)$ *iterations of the while-loop.*

Proof. Fix H at the beginning of an iteration of the while-loop. An *essential component* of H is a connected component of the subgraph of H with vertex set $V(H)$ whose edges are the essential edges in H. Let r denote the number of redundant edges in H and c the number of essential components of H. Fix an optimal tree T in H and a minimal augmentation A for T. The tree T contains exactly $c - 1$ redundant edges of H. Furthermore, if $c > 1$, then each essential component of H is incident with at least 3 redundant edges in H and hence $c \leq 2r/3$. Since the edges of A are essential in $T + A$, less than $2r/3$ edges of $T + A$ are redundant and hence the number of redundant edges goes down by a constant factor greater than 1 in each iteration of the while-loop. The claim follows. □

COROLLARY 1. *Algorithm* 1 *computes a minimal* k*-vertex-connected and a minimal* k*-edge-connected spanning subgraph in* $O(\log n)$ *iterations of the while-loop.*

One drawback of Algorithm 1 is that the redundant edges of H need to be computed at each iteration. In general, it is not clear whether computing these edges is easier than the original problem of finding a minimal spanning P-subgraph. One can avoid this computation by gradually building up a set of essential edges, as shown in the following algorithm. We do not need to assume here that P implies 2-edge-connectivity.

ALGORITHM 2. Computing a minimal spanning P-subgraph of G.
Input P-graph G.
Output Minimal spanning P-subgraph H of G.
(1) $H := G$; $S := \emptyset$;
(2) while $S \neq E(H)$, do:
  (2.1) compute a spanning tree T_H in H with a maximum number of edges in S;
  (2.2) compute a minimal subset A of edges in H such that $T_H + A$ has property P;
  (2.3) $H := T_H + A$; $S := S \cup A \cup \{\text{cut edges in } H\}$.

THEOREM 3. *Algorithm* 2 *computes a minimal spanning* P*-subgraph of* G *for any property* P *satisfying* C1 *and* C2.

Proof. By induction on the number of iterations of the while-loop, one shows that at the end of each iteration H is a spanning P-subgraph of G and the edges in S are essential in H. To prove termination, consider one execution of the while-loop. Let $e \in E(H) - S$. If e is a cutedge in $T_H + A$, then e will be added to S in step 2.3. Otherwise the optimal tree does not contain all edges of $E(H) - S$. At least one edge of $E(H) - S$ will thus be added to S in step 2.3 (as an edge of A) or discarded. In all these cases $|E(H) - S|$ decreases by at least one in this iteration. Termination follows. □

Unlike Algorithm 1, Algorithm 2 is not guaranteed to terminate quickly if P implies 2-edge-connectivity. For instance, if P denotes 2-edge-connectivity and G is a cycle on n vertices, then Algorithm 2 adds exactly one edge to S in each iteration of the while-loop and thus requires n iterations.

We overcome this problem as follows. Let $S \subseteq E(H)$. An edge of H is called *S-critical* if it is one of exactly two edges connecting some connected component of $(V(H), S)$ with the set of vertices outside this component. Note that any S-critical edge is essential in H provided P implies 2-edge-connectivity. Thus, at each iteration we can add to the current set of essential edges the edges of A as well as the S-critical edges in H. The following algorithm makes use of this idea.

ALGORITHM 3. Computing a minimal spanning P-subgraph of G.
Input P-graph G.
Output Minimal spanning P-subgraph H of G.
(1) $H := G$; $S := \emptyset$;
(2) while $S \neq E(H)$, do:
  (2.1) compute a spanning tree T_H in H with a maximum number of edges of S;
  (2.2) compute a minimal $A \subseteq E(H)$ such that $T_H + A$ has property P;
  (2.3) $H := T_H + A$; $B := \{S\text{-critical edges in } H\}$; $S := S \cup A \cup B$.

As before we say that T_H is an optimal tree in H and A is a minimal augmentation for T_H in H.

THEOREM 4. *Algorithm* 3 *computes a minimal spanning* P*-subgraph of* G *for any property* P *satisfying* C1 *and* C2 *and implying* 2*-edge-connectivity.*

Proof. An induction on the iteration number shows that, at the end of each iteration of the while-loop, H is a spanning P-subgraph of H and all edges in S are essential in H. Thus, upon termination H is a minimal spanning P-subgraph of H. To prove termination, fix an iteration of the while-loop. The tree T_H contains a minimum number of edges in $E(H) - S$. Since H is 2-edge-connected, it has no cutedges. Thus there exists a spanning tree in H excluding a single edge of $E(H) - S$. It follows that T_H does not contain all edges of $E(H) - S$. Any edge of $E(H) - S$ that does not belong to T_H is either discarded or added to S in step 2.3. Hence $|E(H) - S|$ is strictly decreasing. Termination follows. □

The proof of Theorem 2 suggests that the number of iterations of Algorithm 3 may be proportional to the number of edges in the input graph. As was the case for Algorithm 1, a much sharper bound holds for properties that imply 2-edge-connectivity.

THEOREM 5. *Assume property* P *implies* 2*-edge-connectivity. Let* C_i *denote* $E(H) - S$ *at the beginning of iteration* i *of the while-loop of Algorithm* 3. *Then,* $|C_{i+1}| \leq 3|C_i|/4$ *for* $|C_i| > 0$. *Thus Algorithm* 3 *terminates after* $O(\log n)$ *iterations.*

Proof. Let H_i and S_i denote H and S at the start of iteration i of the while-loop in Algorithm 3. Hence $C_i = E(H_i) - S_i$. The claim certainly holds if the number of edges of C_i in tree T_{H_i} (computed in step 2.1) is at most $3|C_i|/4$. Now assume that more than $3|C_i|/4$

edges of C_i belong to T_{H_i}. Construct H' from H_i by collapsing the vertex sets of the connected components of $(V(H_i), S_i)$ in H_i (one at a time). Let a be the number of degree-2 vertices in H'. Note that a is a lower bound on the number of edges added to B (and hence to S) in step 2.3. It suffices to show that $a \geq |C_i|/4$.

We have $n(H') > 3|C_i|/4$(*) since we assumed that T_{H_i} contains more than $3|C_i|/4$ edges of C_i. Also $m(H') \leq |C_i|$ and therefore, by inequality (*), $m(H') < 4n(H')/3$. Hence, $\sum_{v \in V(H')} \deg_{H'}(v) < 8n(H')/3$. Moreover, $\sum_{v \in V(H')} \deg_{H'}(v) \geq 2a + 3(n(H') - a)$. Thus $a > n(H')/3$. With inequality (*) we get $a > |C_i|/4$. □

THEOREM 6. *Assume that P satisfies* C1 *and* C2 *and also implies 2-edge-connectivity. If a minimal augmentation of a spanning tree can be computed in time $t(m, n)$, then a minimal spanning P-subgraph can be computed using Algorithm* 3 *in time $O(t(m, n) \log n)$. In particular this holds for k-vertex- and k-edge-connectivity for $k \geq 2$.*

Proof. We claim that all steps in Algorithm 3 other than the minimal augmentation step can be done in linear time and space. We compute an optimal tree T_H as follows. We compute spanning trees T_i for the connected components of $(V(H), S)$. We collapse the connected components of $(V(H), S)$ in H and let T be a spanning time in the resulting graph. The edges in $\cup_i E(T_i) \cup E(T)$ form an optimal tree in H and can be computed in linear time and space using depth-first search. To compute the S-critical edges, we determine the connected components of $(V(H), S)$ in linear time and space using depth-first search. Thus each iteration of the while-loop requires time $O(t(m, n))$. By the previous result Algorithm 3 terminates after $O(\log n)$ iterations of the while-loop. The claim follows. □

In [13] linear time algorithms for computing a minimal augmentation for 2-edge-connectivity and biconnectivity are described. By Theorem 6, Algorithm 3 computes a minimal 2-edge-connected and a minimal biconnected spanning subgraph in time $O((m + n) \log n)$. In fact the following stronger bound applies.

THEOREM 7. *Algorithm* 3 *runs in time $O(m + n \log n)$ time for 2-edge-connectivity and biconnectivity.*

Proof. It suffices to show that the number of edges in the graph H is $O(n)$ after one iteration of the while-loop. We prove this by showing that the number of edges in the minimal augmentation computed in the first iteration is $O(n)$. In the remainder of this proof T_H and A denote the optimal tree and its minimal augmentation computed in the first iteration of the while-loop of Algorithm 3. For 2-edge-connectivity we argue as follows: since every edge of A is essential in $T_H + A$, for every edge $e \in A$ there exists an edge e' in T_H such that e is the only edge in A with the property that e' lies on the (unique) cycle of $T_H + e$. Thus $|A| \leq m(T_H) = n - 1$. For biconnectivity a result of Plummer [19] states that every minimal biconnected graph (i.e., biconnected graph with no redundant edges) has $O(n)$ edges. Thus $|A| = O(n)$ for biconnectivity as well. □

Algorithms for the same problems achieving similar time bounds are presented in [13]. Those algorithms are more complicated than Algorithm 3 because they require redundant edges to be computed explicitly (a fairly involved procedure using ideas from triconnectivity testing). In §5 we shall prove that the bound given in the last theorem is tight. In the next section we describe how to modify Algorithm 3 so that it runs in linear time for 2-edge-connectivity and biconnectivity.

We obtain a result similar to Theorem 6 for the parallel complexity of computing a minimal spanning P-subgraph. (For a definition of the PRAM model see [11].)

THEOREM 8. *Assume that P satisfies* C1 *and* C2 *and also implies 2-edge-connectivity. If a minimal augmentation of a spanning tree can be computed in time $t(m, n)$ on $p(m, n)$ PRAM processors, than a minimal spanning P-subgraph can be computed using algorithm* 3 *in time*

$O((t(m,n)+c(m,n))\log n)$ on $p(m,n)$ processors, where $c(m,n)$ is the time required by steps 2.1 *and* 2.3 *of algorithm* 3 *on $p(m,n)$ processors.*

We note that on ARBITRARY PRAM the complexity of steps 2.1 and 2.3 is dominated by that of finding connected components, i.e., they can be performed with almost optimal speedup in time $O(\log n)$ using $O((m+n)\alpha(m,n)/\log n)$ processors [4], where α denotes the inverse Ackermann function. The minimal augmentation step can be done in polylogarithmic time on a linear number of processors, as shown in [13] for biconnectivity and 2-edge-connectivity.

4. Linear time algorithms. In this section we adapt Algorithm 3 to compute minimal spanning subgraphs for 2-edge-connectivity and biconnectivity in linear time. The linear time bound is achieved by combining the linear time minimal augmentation procedures given in [13] with a method for reducing the size of the current graph while preserving its 2-edge-connectivity (biconnectivity) structure.

4.1. Finding a minimal 2-edge-connected spanning subgraph. We start with a description of two graph operations that preserve the 2-edge-connectivity structure. Let H be a graph that may not be 2-edge-connected and let $S \subseteq E(H)$. An *S-component of H* is a 2-edge-connected component of $(V(H), S)$. The operation of *shrinking an S-component of H* consists of collapsing the vertex set of this S-component in H. For the second operation define a *chain in H* to be a path in H of length at least 2 whose internal vertices all have degree 2 in H. Note that the edges in a chain are essential in H. A chain is *maximal* if it is not a proper subgraph of another chain in H. The operation of *contracting a chain in H* consists of collapsing the set of internal nodes of the chain in H.

If graph Q is obtained from graph H by shrinking all S-components in H and contracting all maximal chains in the resulting graph, we say that Q is a *full contraction of H* with respect to S. The following algorithm is a variant of Algorithm 3 in which, at the end of the while-loop, H is replaced by its full contraction. In step 2.0 we replace H by a sparse 2-edge-connected subgraph to speed up subsequent steps; this also simplifies the analysis of Algorithm 3. A linear time algorithm for computing a minimal augmentation for 2-edge-connectivity (step 2.2) is given in [13]. We finally note that the intermediate graphs need not be simple even if the input graph G is simple.

ALGORITHM 4. Computing a minimal 2-edge-connected spanning subgraph of G.
Input 2-edge-connected graph G.
Output Minimal 2-edge-connected spanning subgraph of G.
(1) $H := G$; $S := \emptyset$;
(2) while $E(H) \not\subseteq S$, do:
  (2.0) replace H by an ear decomposition of H; discard the edges in the trivial ears from H;
  (2.1) compute a spanning tree T_H in H with a maximum number of edges of S;
  (2.2) compute a minimal $A \subseteq E(H)$ such that $T_H + A$ is 2-edge-connected;
  (2.3) $H := T_H + A$; $B := \{S\text{-critical edges in } H\}$; $S := S \cup A \cup B$;
  (2.4) replace H by its full contraction with respect to $S \cap E(H)$;
(3) return graph $(V(G), S)$.

To prove the correctness of Algorithm 3, we first need to establish that the two operations of shrinking S-components and contracting chains preserve the 2-edge-connectivity structure of H.

LEMMA 1. *Let H be a graph that may not be 2-edge-connected and let $S \subseteq E(H)$. If H' is obtained from H by contracting a chain or shrinking an S-component, then H' is 2-edge-connected if and only if H is 2-edge-connected. Thus, a full contraction of H is 2-edge-connected if and only if H is 2-edge-connected.*

Proof. The claim for a full contraction follows with a simple inductive argument from the first claim. We now prove the first claim. Let H' be obtained from a 2-edge-connected graph H by contracting a chain P. A cycle C in H containing an edge e of $E(H')(\subseteq E(H))$ either contains all edges of P or none. In the latter case C is also a cycle in H' containing e. In the former case we obtain a cycle C' in H' containing e by contracting chain P on cycle C. Now suppose that H' is obtained from 2-edge-connected graph H by collapsing vertex set $X \subseteq V(H)$ of an S-component. An edge $e \in E(H')$ lies on a cycle C in H. Let P be a maximal subpath of C containing e and not having a vertex of X as an internal vertex. The edges on P form a cycle in H' containing e. Since H' is also connected in both cases, we conclude that H' is indeed 2-edge-connected.

Now assume that H' is 2-edge-connected and obtained from H by contracting a chain or shrinking an S-component. Again we see that H is connected. Next we note that a cycle C' in H' yields a cycle C in H that includes all edges on C (and possibly other edges). Since every edge of H' lies on a cycle of H', every edge in $E(H')$ lies on a cycle in H. If $e \in E(H) - E(H')$, then either e connects two vertices in the same S-component of H or it lies on a chain in H. In the first case it lies on a cycle whose vertices belong to the S-component. In the second case some other edge on the same chain in H belongs to $E(H')$ and thus lies on a cycle of H; this cycle must also include e. We conclude that H is 2-edge-connected. □

COROLLARY 2. *Let H be a 2-edge-connected graph and let H' be obtained from H by contracting a chain or shrinking an S-component ($S \subseteq E(H)$). An edge $e \in E(H')$ is essential in H' if and only if it is essential in H. The same claim holds if H' is a full contraction of H.*

Proof. As before, the claim for the full contraction follows by a straightforward induction from the first claim. To prove the first claim, fix $e \in E(H')$. If H' is obtained from H by shrinking an S-component, then $H' - e$ is obtained from $H - e$ by shrinking the same S-component since e does not belong to that S-component. The claim of the corollary follows with the previous lemma. If H' is obtained from H by contracting a chain, then e is essential in both H and H' if it belongs to a chain in H' (and thus belongs to a chain in H). Otherwise $H' - e$ is obtained from $H - e$ by contracting a chain and the previous lemma implies the claim of the corollary. □

THEOREM 9. *Algorithm 4 outputs a minimal 2-edge-connected spanning subgraph of G.*

Proof. Replacing H by its full contraction in step 2.4 does not increase $|E(H) - S|$. As in the proof of Theorem 4, one argues that $|E(H) - S|$ decreases by at least 1 during one iteration of the while-loop. The termination of Algorithm 4 follows.

We number the iterations of the while-loop from 0 to k (in iteration k Algorithm 4 finds that $E(H) \subseteq S$). Let H_i and S_i denote the graph H and the set S at the start of the ith iteration, let T_i and A_i stand for T_H and A in iteration i for $i < k$, and let H_i' denote the graph $(V(H_i), S_k \cap E(H_i))$. Lemma 1 implies that each H_i is 2-edge-connected. It suffices to prove that H_i' is a minimal 2-edge-connected spanning subgraph of H_i for $0 \le i \le k$. The claim of the theorem then follows since $H_0 = G$ and H_0' is returned in step 3 of Algorithm 4.

To prove the claim, we need the following fact:

(*) any edge in $E(H_i) \cap S_i$ is essential in H_i for $0 \le i \le k$.

We prove (*) by induction on i. Since $S_0 = \emptyset$, (*) holds for $i = 0$. Suppose it holds for $i = j$ where $j < k$. Since $E(H_{j+1}) \subseteq E(H_j)$, every edge in $E(H_{j+1}) \cap S_j$ is essential in H_j and hence essential in $T_j + A_j$. Since H_{j+1} is a full contraction of $T_j + A_j$, Corollary 2 implies that these edges are essential in H_{j+1} as well. Furthermore, the edges in $E(H_{j+1}) \cap (S_{j+1} - S_j)$ are essential in $T_j + A_j$ and thus essential in H_{j+1} (by Corollary 2). Claim (*) follows.

We now prove that each H_i' is a minimal 2-edge-connected spanning subgraph of H_i by induction on $k - i$. For the base case $i = k$ we first note that $H_k = H_k'$ is 2-edge-connected.

With (*) it also follows that all edges in H_k are essential since $E(H_k) \cap S_k = E(H_k)$. This completes the proof of the base case.

For the induction step assume that H'_{i+1} is a minimal 2-edge-connected spanning subgraph of H_{i+1}. In particular H'_{i+1} is 2-edge-connected. By Lemma 1 it suffices to show that H'_{i+1} is a full contraction of H'_i in order to establish that H'_i is 2-edge-connected. Since the edges in $E(H_i) - E(T_i + A_i)$ are redundant in H_i, (*) implies that they do not belong to S_i and hence do not belong to S_k. Thus $E(H_i) \cap S_k = E(T_i + A_i) \cap S_k$. The fact that H_{i+1} is a full contraction of $T_i + A_i$ (with respect to a subset of S_k) implies that H'_{i+1} is obtained from $(V(H_i), E(T_i + A_i) \cap S_k)$ by shrinking components and contracting chains. But the latter graph is simply H'_i. Thus H'_i is 2-edge-connected. Since each edge in H'_{i+1} is essential and H'_{i+1} is a full contraction of H'_i, each edge of H'_{i+1} is essential in H'_i by Corollary 2. Furthermore each edge in $E(H'_i) - E(H'_{i+1})$ either belongs to a chain in $T_i + A_i$ or belongs to $S_k \cap E(T_i + A_i)$. In the former case the edge is essential in H'_i because it has an endpoint of degree 2. In the latter case it belongs either to S_i, in which case it is essential by (*), or to $S_{i+1} - S_i$, in which case it is essential in $T_i + A_i$ and hence essential in H'_i. We conclude that H'_i is a minimal 2-edge-connected spanning subgraph of H_i. □

The following technical lemma will be used in the analysis of Algorithm 4.

LEMMA 2. *Let F be a forest in which r nodes are marked. Let $n_i(F)$ denote the number of degree i nodes in F. If every chain in F that does not contain a marked node as an internal vertex has length at most k, then $n(F) < n_0(F) + k(2n_1(F) + r)$.*

Proof. An *unmarked chain* in F is a chain that does not contain a marked vertex as an internal node. Construct F' by contracting all maximal unmarked chains of F into single edges. Assume that all nodes in F and hence all nodes in F' have degree at least 1. Let n_1, n_2, and n_3 denote the number of nodes of degree 1, degree 2, and degree ≥ 3, respectively, in F'. We know that $\sum_{v \in V(F')} \deg_{F'}(v) \leq 2n(F') - 2$. Since the left-hand side is at least $n_1 + 2n_2 + 3n_3$ and $n(F') = n_1 + n_2 + n_3$, we find that $n_3 \leq n_1 - 2$ and hence $n_3 < n_1$. By the definition of F' we have $n_2 \leq r$ and thus $n(F') < 2n_1 + r$. By noting that $n(F) \leq n(F') + (k-1)m(F')$ and hence $n(F) < kn(F')$, it follows that $n(F) < k(2n_1 + r)$. Since $n_1 = n_1(F)$, the claim of the lemma follows. □

LEMMA 3. *Let H be a 2-edge-connected graph and let S be a proper subset of $E(H)$. Let Q be a full contraction of H with respect to S. Then $n(Q) < 13|E(H) - S|$.*

Proof. Let C denote the set $E(H) - S$ and let Q' denote the graph $(V(Q), S \cap E(Q))$. The graph Q' is a forest. We may assume that $n(Q) > 1$ (otherwise the claim is trivial). Since Q is 2-edge-connected, each node of degree 0 in Q' is incident in Q with at least 2 edges of C (since Q is 2-edge-connected) and each node of degree 1 in Q' is incident in Q with at least one edge of C. Thus, Q' has at most $|C|$ nodes of degree 0 and at most $2|C|$ nodes of degree 1. Mark each endpoint of an edge of C in Q'. Each unmarked chain in Q' has length at most 2. By applying Lemma 2 to Q', we get $n(Q') < |C| + 2(4|C| + 2|C|)$ and hence $n(Q) = n(Q') < 13|C|$ as claimed. □

COROLLARY 3. *Let k denote the number of iterations of the while-loop of Algorithm 4. Let H_i denote the graph H at the start of the ith iteration of the while-loop. Then $m(H_{i+14}) < .83m(H_i)$ for $i + 14 \leq k$.*

Proof. First, we show that

$$m(H_{i+1}) < 2n(H_i) \tag{1}$$

for $i < k$. To see this, let H' be the subgraph of H_i consisting of the nontrivial ears in the ear decomposition found in step 2.0 of Algorithm 4 and let q denote the number of those ears. Then $m(H') \leq n(H') + q$ and $q < n(H')$, hence $m(H') < 2n(H')$. Since $n(H') = n(H_i)$ and $m(H_{i+1}) \leq m(H')$, inequality (1) follows.

Let q_i denote $|E(H) - S|$ at the start of the ith iteration of the while-loop of Algorithm 4. By Theorem 5 we have $q_{i+1} \leq 3q_i/4$ for $i < k$ and hence $q_{i+j} \leq (3/4)^j q_i$ for $i + j \leq k$. With Lemma 3 we obtain

$$n(H_{i+j+1}) < 13q_{i+j} \leq 13(3/4)^j q_i \leq 13(3/4)^j m(H_i) \tag{2}$$

for $i + j + 1 \leq k$. Combining (1) and (2) yields

$$m(H_{i+j+2}) < 26(3/4)^j m(H_i) \tag{3}$$

for $i + j + 2 \leq k$. Substituting $j = 12$ yields the claim. □

THEOREM 10. *Algorithm* 4 *computes a minimal* 2*-edge-connected spanning subgraph of any* 2*-edge-connected graph on* n *vertices and* m *edges in time and space* $O(n + m)$.

Proof. Consider one iteration of the while-loop. We compute an ear decomposition in linear time and space using the algorithm of [21]. We compute an optimal tree T_H and S-critical edges in linear time and space as described in the proof of Theorem 6. We compute a minimal augmentation in linear time and space using the algorithm of [13]. To compute the S-critical edges, we determine the connected components of $(V(H), S)$. Finally, the complexity of computing a full contraction is dominated by the complexity of computing 2-edge-connected components [22], i.e., it can be done in linear time and space. In summary, one iteration of Algorithm 4 can be performed in linear time and space. The claim follows with Corollary 3. □

Note. An efficient *NC* algorithm for this problem is given in [13]. With Theorem 10 the work of this algorithm (time-processor product) can be reduced by a factor of $\Theta(\log n)$ using standard techniques.

4.2. Finding a minimal biconnected spanning subgraph. The linear time algorithm for computing a minimal 2-edge-connected spanning subgraph makes use of the two operations of shrinking S-components and contracting chains. In this section we exhibit a pair of operations on biconnected graphs with similar properties, although they are more complicated. They will be used in the linear time algorithm for computing a minimal biconnected spanning subgraph.

Let H be biconnected and $S \subseteq E(H)$. An *S-block* of H is a block of the graph $(V(H), S)$. The graph $(V(H), S)$ need not be connected. Thus, an S-block (or essential block) of H is either an isolated vertex in $(V(H), S)$, a cutedge in $(V(H), S)$, or a maximal biconnected subgraph of $(V(H), S)$ with at least 3 vertices.

Let B be an S-block of H with at least 3 vertices. An *internal vertex* of B is a vertex in B that is neither a cutpoint in $(V(H), S)$ nor is it incident in H with an edge of $E(H) - S$; we write $I(B)$ for the set of internal vertices of B. The operation of *shrinking the S-block B* in H consists of deleting all edges of B in H as well as all internal vertices of B, connecting the remaining vertices of B into a cycle in arbitrary order, and subdividing each edge of this cycle with a new vertex. Thus, if $u_1, \ldots, u_k$ are the noninternal vertices of B, and $V' = \{v_1, \ldots, v_k\}$ is the set of k new vertices used to subdivide the edges of the cycle, then the resulting graph has vertex set $(V(H) - I(B)) \cup V'$ and edge set $(E(H) - E(B)) \cup C_B$ where $C_B = \{(u_1, v_1), (v_1, u_2), \ldots, (u_{k-1}, v_{k-1}), (v_{k-1}, u_k), (u_k, v_k), (v_k, u_1)\}$. The operation is illustrated in Fig. 1.

The following results establish that shrinking S-blocks preserves the biconnectivity structure of a graph.

LEMMA 4. *Let H be a graph that may not be biconnected and let $S \subseteq E(H)$. Construct H' from H by shrinking an S-block B in H. Then H' is biconnected if and only if H is biconnected.*

Proof. Assume that H is biconnected. Thus, for any three distinct vertices u, v, w in H there exists a path from u to w avoiding v. Now suppose that H' is not biconnected. Let C_B

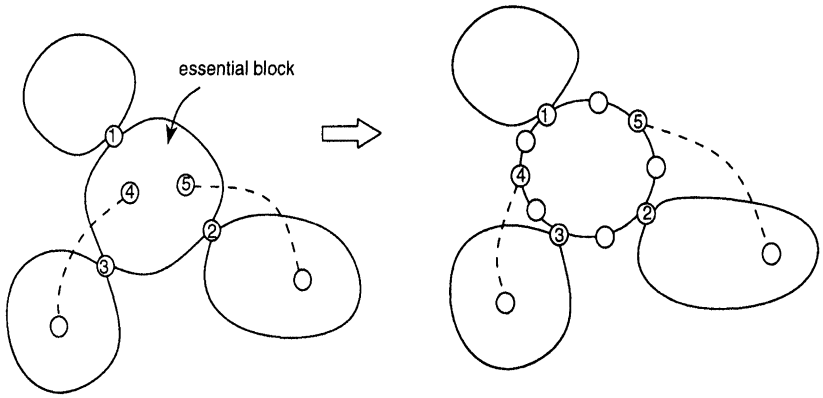


FIG. 1. *Shrinking the essential block containing vertices numbered* 1–5. *The dashed edges are in* $E(H) - S$.

denote the cycle in H' that corresponds to S-block B in H. For some distinct $u, v, w \in V(H')$ every path from u to w in H' passes through v. We may assume that $u, v, w \in V(H)$. To see this, note that if a vertex in $\{u, v, w\}$ does not belong to H (i.e., it is a vertex of degree 2 on C_B), then we may replace it by one of its neighbors on C_B while maintaining the property that any path from u to w goes through v. If $\{u, v, w\} \subseteq V(H)$, then there exists a path P in H from u to w avoiding v. From P we obtain a path P' in H' from u to w by replacing each edge (z, z') of B on P by a path from z to z' on C_B avoiding v. Clearly P' avoids v. We conclude that H' is biconnected.

Now suppose that H' is biconnected. To show that H is biconnected, we need to prove that for any three distinct vertices u, v, and w in H there is a path from u to w in H avoiding v. Since H' is biconnected, there exists a path P from u to w avoiding v in H'. If P contains a subpath from u_i to u_j on C_B, then we can replace each such subpath with a path from u_i to u_j in B avoiding v (since B is biconnected). We thus obtain a path P' from u to w in H that avoids v. We conclude that H is biconnected. □

COROLLARY 4. *Let H be biconnected and let H' be obtained from H by shrinking an S-block B in H. An edge $e \in E(H') \cap E(H)$ is essential in H' if and only if it is essential in H.*

Proof. The edge e does not belong to B. Thus B is an S-block in $H - e$. The graph $H' - e$ may be obtained from $H - e$ by shrinking S-block B in $H - e$ and possibly subdividing edges on the cycle C_B that replaces the block B of $H - e$. (The subdivisions are only necessary if B has more internal vertices in $H - e$ than it has in H. This may happen if e is the only edge in $E(H) - S$ incident on some vertex in B or e is the only edge in S incident on a cutpoint B.) Since the biconnectivity property is closed under subdivisions, the previous lemma implies that $H - e$ is biconnected if and only if $H' - e$ is biconnected. □

As before let H be a biconnected graph and $S \subseteq E(H)$. The second operation on biconnected graphs is defined on the block structure of $(V(H), S)$. An *S-block chain* in H is an alternating sequence $c_1 B_1 \ldots c_k B_k c_{k+1}, k > 1$, of vertices and S-blocks in H with the following properties: (i) each B_i $(1 \leq i \leq k)$ has exactly two cutpoints in $(V(H), S)$, namely c_i and c_{i+1}; (ii) for $1 < i < k$, B_i intersects exactly two blocks, namely B_{i-1} and B_{i+1} in c_i and c_{i+1}, respectively; (iii) no vertex in any B_i except possibly c_1 and c_{k+1} is incident with an edge not lying in any B_i. A *maximal S-block chain* in H is a block chain in H not properly contained in any other S-block chain of H. If the set S is understood, we may refer to an S-block chain as a block chain. Figure 2 shows a maximal S-block chain $c_1 B_1 c_2 B_2 c_3 B_3 c_4 B_4 c_5$.

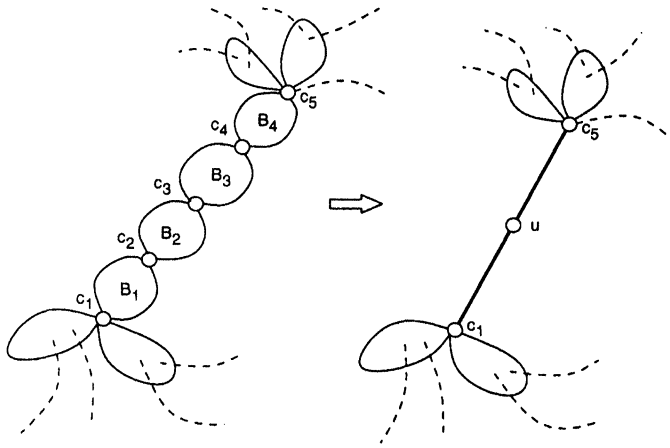


FIG. 2. *Contracting block chain* $c_1B_1c_2B_2c_3B_3c_4B_4c_5$. *The dashed lines represent edges in* $E(H) - S$.

The operation of *contracting the S-block chain* $c_1B_1 \ldots c_kB_kc_{k+1}$ in H consists of deleting in H all vertices in the S-blocks of this sequence except c_1 and c_{k+1} and adding a new vertex z and two new edges (c_1, z) and (z, c_{k+1}) (see Fig. 2).

LEMMA 5. *Let H be a graph that may not be biconnected and $S \subseteq E(H)$. Construct H' by contracting the S-block chain $c_1B_1 \ldots c_kB_kc_{k+1}$ in H. Then H' is biconnected if and only if H is biconnected.*

Proof. Assume that H is biconnected. If H' is not biconnected, then there exist distinct $u, v, w \in V(H')$ such that any path from u to w in H' passes through v. Since H is biconnected, there is a path from c_1 to c_{k+1} in H avoiding c_2, yielding a path in H' from c_1 to c_{k+1} avoiding the new vertex z. Therefore $v \neq z$. If $u = z$ or $w = z$, then we can replace either vertex by c_1 or c_{k+1} while maintaining the property that any path from u to w in H' passes through v. We may thus assume that $u, v, w \neq z$ and hence $u, v, w \in V(H) \cap V(H')$. Thus there is a path P in H from u to w avoiding v. If P does not traverse an edge in any B_i, then it is a path in H' from u to w avoiding v. Otherwise P contains a subpath from c_1 to c_{k+1} (c_{k+1} to c_1). We may replace this subpath by the path $\langle c_1, z, c_{k+1}\rangle(\langle c_{k+1}, z, c_1\rangle)$ in H', thus obtaining a path in H' from u to w that avoids v. Hence H' is biconnected.

Now let H' be biconnected. If H is not biconnected, then for some distinct vertices u, v, w in H any path from u to w in H goes through v. Since H' is biconnected, there is a path in H' from c_1 to c_{k+1} avoiding z, yielding a path in H from c_1 to c_{k+1} for which no internal vertices belong to any B_i. It follows that v is not a vertex in any B_i other than c_1 or c_{k+1}. It also implies that we may assume that neither u nor w is a vertex in any B_i other than c_1 or c_{k+1} while maintaining the property that any path from u to w in H passes through v. Thus the path in H' from u to w avoiding v yields a path in H from u to w avoiding v. Therefore H is biconnected. □

COROLLARY 5. *Let H be biconnected and let H' be obtained from H by contracting a block chain $c_1B_1 \ldots c_kB_kc_{k+1}$ in H. An edge $e \in E(H') \cap E(H)$ is essential in H' if and only if it is essential in H.*

Proof. Since e belongs to H and H', it does not belong to any block B_i in the block chain. Thus $c_1B_1 \ldots c_kB_kc_{k+1}$ is a block chain in $H - e$ and $H' - e$ is obtained from $H - e$ by contracting this block chain in $H - e$. By the previous lemma $H - e$ is biconnected if and only if $H' - e$ is biconnected. The claim follows. □

If the graph Q is obtained from H by first shrinking all S-blocks of H and then contracting all maximal block chains in the resulting graph, we say that Q is a *full contraction* of H with respect to S. The following algorithm is a variation of Algorithm 3 in which we replace H by its full contraction at the end of each iteration of the while-loop. It is shown in [13] that the minimal augmentation step (step 2.2) can be done in linear time. We note that the intermediate graph H will always be simple assuming that the input graph G is simple (the usual assumption for biconnected graphs).

ALGORITHM 5. Computing a minimal biconnected spanning subgraph of G.
Input Biconnected graph G.
Output Minimal biconnected spanning subgraph of G.
(1) $H := G$; $S := \emptyset$;
(2) while $E(H) \not\subseteq S$, do:
  (2.0) replace H by an open ear decomposition of H; remove edges in trivial ears from H;
  (2.1) compute a spanning tree T_H in H with a maximum number of edges of S;
  (2.2) compute a minimal $A \subseteq E(H)$ such that $T_H + A$ is biconnected;
  (2.3) $H := T_H + A$; $B := \{S\text{-critical edges in } H\}$; $S := S \cup A \cup B$;
  (2.4) replace H by its full contraction with respect to $S \cap E(H)$; add new edges to S;
(3) return graph $(V(G), E(G) \cap S)$.

THEOREM 11. *Algorithm 5 outputs a minimal biconnected spanning subgraph of G.*

Proof. The proof is very similar to the proof of Theorem 9. The termination of the algorithm is established by showing that $|E(H) - S|$ is strictly decreasing exactly as in the proof of Theorem 9. Note that newly created edges are both added to $E(H)$ and S and thus do not increase $|E(H) - S|$.

Reusing the notation of that proof, we need to show that H_i' is a minimal biconnected spanning subgraph of H_i. The statement (*) that any edge in $E(H_i) \cap S_i$ is essential in H_i is still true. The proof proceeds by induction over j just as in the proof of Theorem 9 with the following minor modification: the edges in $E(H_{j+1}) \cap (S_{j+1} - S_j)$ are either essential in $T_j + A_j$ (as before) or they are new edges added by the full contraction. The edges in the first class are essential in H_{j+1} by Corollaries 4 and 5 since H_{j+1} is a full contraction of $T_j + A_j$. The edges in the second class (new edges in H_{j+1}) are essential in H_{j+1} since they have an endpoint of degree 2.

One now proves that H_i' is a minimal biconnected spanning subgraph of H_i as in the proof of Theorem 9. The main difference is in the inductive argument that H_i' is a minimal biconnected spanning subgraph of H_i: the fact that H_{i+1} is a full contraction of $T_i + A_i$ with respect to some $X \subseteq S_k$ implies that $H_{i+1}' = (V(H_{i+1}), E(H_{i+1}) \cap S_k)$ is obtained from $(V(T_i + A_i), E(T_i + A_i) \cap S_k) = H_i'$ by shrinking X-blocks and contracting X-block chains possibly followed by subdividing edges. With Lemma 4 and Lemma 5, this implies that H_i' is 2-edge-connected. By Corollaries 4 and 5 (and the induction assumption) the edges in $E(H_i') \cap E(H_{i+1}')$ are essential in H_i'. The edges in $E(H_i') - E(H_{i+1}')$ belong either to S_i, in which case they are essential in H_i and hence in H_i' by (*), or to $S_{i+1} - S_i$, in which case they are essential in $T_i + A_i$ and hence in H_i'.

The remainder of the proof is the same. □

The following result is needed for the analysis of Algorithm 5.

LEMMA 6. *Let H be biconnected and let S be a proper subset of $E(H)$. If Q is a full contraction of H with respect to S, then $n(Q) < 60|E(H) - S|$.*

Proof. Let C denote the set $E(H) - S$ and let Q' denote the graph $Q - C$. To bound $n(Q)$, we consider the block graph of Q'. Let H' be an arbitrary graph. The *block graph* of H' [23], denoted by $\text{blk}(H')$, is a bipartite graph whose vertices are the cutpoints and blocks of H'. A block is connected in $\text{blk}(H')$ to exactly those cutpoints that it contains in H'. It is known [23] that the block graph of H' is a tree for any connected graph H'. Thus the graph $\text{blk}(Q')$ is a forest.

Let n_0 and n_1 denote the number of degree 0 and degree 1 nodes in $\text{blk}(Q')$. If $n(\text{blk}(Q')) = 1$, then the graph $(V(H), S)$ has a unique block and this block contains at least 3 vertices. Hence Q' is a cycle (obtained by shrinking this block) of even length. Every other vertex on this cycle is incident with an edge of $C(\neq \emptyset)$. Hence $n(Q) \leq 4|C|$. If $n(\text{blk}(Q')) > 1$, then each node of degree 0 in $\text{blk}(Q')$ represents a block in Q' that is incident with at least 2 edges of C in Q and each node of degree 1 in $\text{blk}(Q')$ represents a block in Q' that is incident with at least one edge of C in Q. Thus $n_0 \leq |C|$ and $n_1 \leq 2|C|$. Mark a vertex in $\text{blk}(Q')$ representing a cutpoint in Q if it is incident in Q with an edge of C and mark a vertex in $\text{blk}(Q')$ representing a block of Q' if a vertex in this block other than a cutpoint is incident with an edge of C. From the definition of a full contraction it follows that any chain of unmarked nodes in $\text{blk}(Q')$ has length at most 6. By applying Lemma 2 we see that $\text{blk}(Q')$ has fewer than $|C| + 6(4|C| + 2|C|) = 37|C|$ vertices. Since there are more blocks in Q' than there are cutpoints, Q' has fewer than $19 \cdot |C|$ cutpoints.

We partition the vertices of Q' into 3 classes: class 1 contains the cutpoints in Q', class 2 includes the endpoints of edges of C (that are not cutpoints), and class 3 comprises the new vertices used to subdivide cycles when shrinking S-blocks in H. Let p_1, p_2, and p_3 denote the number of vertices in class 1, class 2, and class 3, respectively. Clearly $p_2 \leq 2|C|$. Above we have shown that $p_1 < 19|C|$. Note that the number of class 3 vertices in any block of Q' is no larger than the number of vertices in that block that belong to class 1 or class 2. The sum of the latter number, taken over all blocks of Q', is an upper bound on p_3. This sum is at most $2|C| + m(\text{blk}(Q'))$ and hence $p_3 < 2|C| + 37|C| = 39|C|$. Altogether we find that $p_1 + p_2 + p_3 < 60|C|$ and hence $n(Q') = n(Q) < 60|C|$. □

COROLLARY 6. *Let k denote the number of iterations of the while-loop of Algorithm 5. Let H_i denote the graph H at the start of the ith iteration of the while-loop. Then $m(H_{i+22}) < .77m(H_i)$ for $i + 22 \leq k$.*

Proof. We first argue that

$$m(H_{i+1}) < 4n(H_i) \tag{4}$$

for any $i < k$. Let H' denote the subgraph of H_i consisting of the nontrivial ears in the open ear decomposition found in step 2.0 of Algorithm 5. Just as in the proof of Corollary 3 we have $m(H') < 2n(H') = 2n(H_i)$. If we shrink a block in $T_i + A_i$, the number of new edges added is bounded by the number of vertices in the block, which is at most the number of edges in the block. It follows that the total increase in the number of edges during the full contraction of $T_i + A_i$ is at most $m(T_i + A_i) \leq m(H')$. We conclude that $m(H_{i+1}) < 4n(H_i)$.

Let q_i denote $|E(H) - S|$ at the start of the ith iteration of the while-loop of Algorithm 5. Because any new edge created during the full contraction is also added to S, Theorem 5 implies that $q_{i+1} \leq 3q_i/4$ for $i < k$ and hence $q_{i+j} \leq (3/4)^j q_i$ for $i + j \leq k$. With Lemma 6 we obtain

$$n(H_{i+j+1}) < 60q_{i+j} \leq 60(3/4)^j m(H_i) \tag{5}$$

for $j > 0$ and $i + j + 1 \leq k$. Combining (4) and (5) yields

$$m(H_{i+j+2}) < 240(3/4)^j m(H_i) \tag{6}$$

for $j > 0$ and $i + j + 2 \leq k$. Substituting $j = 20$ yields the claim of the corollary. □

THEOREM 12. *Algorithm 5 finds a minimal biconnected spanning subgraph of any biconnected graph on n vertices and m edges in time and space* $O(n + m)$.

Proof. Consider one iteration of the while-loop. We compute an open ear decomposition in linear time and space using the algorithm of [21]. We compute an optimal tree T_H and S-critical edges in linear time and space as described in the proof of Theorem 6. We compute a minimal augmentation in linear time and space using the algorithm of [13]. Finally, the complexity of computing a full contraction is dominated by the complexity of computing blocks, i.e., it can be done in linear time and space [22]. In summary, one iteration of Algorithm 5 can be performed in linear time and space. The claim follows with Corollary 6. □

As was the case for 2-edge-connectivity the last result yields an improvement by a $\Theta(\log n)$ factor in the work of an efficient NC algorithm described in [13] for computing a minimal biconnected spanning subgraph.

5. Worst-case analysis. In this section we prove that Algorithm 3 requires $\Theta(m+n \log n)$ in the worst case for 2-edge-connectivity and biconnectivity. This result justifies the work we invested in the last section to achieve a linear running time. Because of its simple structure Algorithm 1 will be a more convenient vehicle for proving lower bounds. Fortunately, any lower bound on the number of iterations of Algorithm 1 yields the same lower bound for the number of iterations of Algorithm 3.

LEMMA 7. *Let H be a P-graph, S a subset of the essential edges in H, T a spanning tree in H with a maximum number of essential edges, and A a minimal augmentation for T in H. Then T + A can be rewritten as T′ + A′ where T′ is a spanning tree in H with a maximum number of edges of S and A′ is a minimal augmentation for T′ in H.*

Proof. An *essential component* of H is a maximal subgraph of H containing a spanning tree of essential edges. Let $C_1, \ldots, C_k$ be the essential components of H. For each i let T_i be a spanning tree of essential edges for C_i with a maximum number of edges of S. Let F be the set of edges of T that are redundant in H. Then the tree T' with edge set $F \cup E(T_1) \cup \cdots \cup E(T_k)$ is a spanning tree in H. The tree T' contains a maximum number of edges of S since the intersection of any other tree with C_i is a forest that contains no more than $|E(T') \cap E(C_i)|$ edges of S. We have $E(T') \subseteq E(T + A)$. Let $A' = E(T + A) - E(T')$. The graph $T' + A'(= T + A)$ has property P. Moreover, each edge in A' is essential in $T' + A'$ since T' contains all edges of T that are redundant in H. We conclude that A' is a minimal augmentation for T' in H. □

A *trace of Algorithm* 1 for P-graph G is a sequence of graphs representing H at the start of successive iterations of Algorithm 1. Formally, $H_0, H_1, \ldots, H_l$ is a trace of Algorithm 1 for P-graph G if $H_0 = G$, $H_i \neq H_{i+1}$ for $0 \leq i < l$, and $H_i (0 < i \leq l)$ is of the form $T + A$ where T is an optimal tree in H_{i-1} and A is a minimal augmentation for T in H_{i-1}. The integer l is the *length* of the trace. Similarly, we define a *trace* of Algorithm 3 for P-graph G to be a sequence of graphs representing H at the start of successive iterations of Algorithm 3. The sequence $H_0, H_1, \ldots, H_l$ is a trace of Algorithm 3 for P-graph G if $H_0 = G$, $H_i \neq H_{i+1}$ for $0 \leq i < l$, and, for any sequence of sets $E_0, E_1, \ldots, E_{l-1}$ such that E_i is a set of essential edges in H_i for $0 \leq i \leq l - 1$, we can write H_i as $T_{i-1} + A_{i-1}$ where T_{i-1} is a spanning tree in H_{i-1} containing a maximum number of edges of E_{i-1} and A_{i-1} is a minimal augmentation for T_{i-1} in H_{i-1}. The following result is an immediate corollary of Lemma 7.

COROLLARY 7. *A trace of Algorithm* 1 *for P-graph G is also a trace of Algorithm* 3 *for G.*

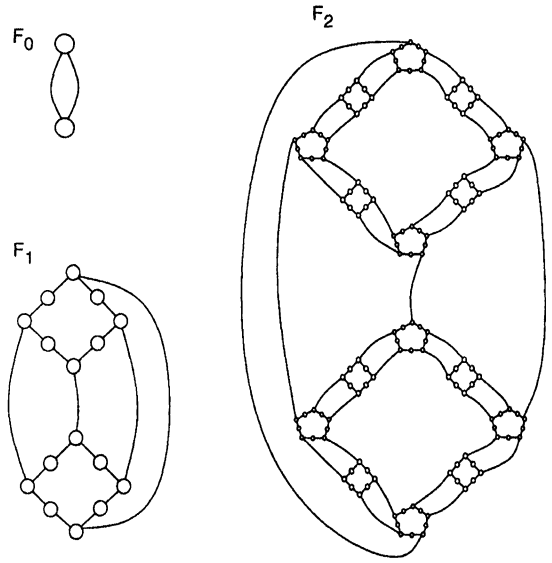


FIG. 3. F_0, F_1, *and* F_2.

By Corollary 7 a lower bound on the number of iterations of Algorithm 1 implies the same lower bound for the number of iterations of Algorithm 3. Below we make use of this fact: we derive a tight $\Theta(\log n)$ lower bound on the (worst-case) number of iterations of Algorithm 1 that also applies to Algorithm 3.

To capture the worst-case behavior of Algorithm 1, we define the *P-complexity* of a graph. Informally, the P-complexity of a P-graph H is the maximum number of iterations that Algorithm 1 may need in order to compute a minimal spanning P-subgraph of H. Formally, we define the P-complexity of H to be the maximum length of a trace of Algorithm 1 for H. If the property P is clear, we shall use the term "complexity" instead of "P-complexity." We denote the P-complexity of graph H by $c_P(H)$, or $c(H)$ if property P is understood. Note that the notion of P-complexity corresponds to an adversary choosing, for a given input graph, an optimal tree and a minimal augmentation in each iteration of the while-loop of Algorithm 1 with the goal of maximizing the number of such iterations. Hence we consider the worst-case behavior of Algorithm 1 not only with respect to different input instances but also with respect to different choices of the algorithm (i.e., sequences of trees and augmentations). We call an infinite sequence of graphs $H_0, H_1, H_2, \ldots$ such that $c_P(H_i) \geq i$ for all $i \geq 0$ a *sample* for P.

We construct a sample for 2-edge-connectivity inductively: F_0 has two vertices with two parallel edges between them. We construct F_i from F_{i-1} as follows: we double each essential edge in F_{i-1}; we call the resulting graph F_i'. For each vertex u of degree d in F_i', we add d new vertices $u_1, u_2, \ldots, u_d$ to F_i and connect them in a cycle. We number the edges incident on each vertex in F_i' as $0, 1, \ldots$ and, for each edge $e = (u, v)$ in F_i' that is the ith edge incident on u and the jth edge incident on v, we add an edge (u_i, v_j) to F_i. Finally, we subdivide each edge of F_i' connecting two vertices u_i, u_j corresponding to the same vertex u in F_i' with a new vertex. Note that F_i is not uniquely defined. The following claims hold for an arbitrary such sequence. Figure 3 shows the first three graphs in a possible sequence.

The proof that the F_i's form a sample for 2-edge-connectivity relies on the following property of 2-edge-connectivity. Let us say that H' is an essential contraction of H if H' is obtained from H by shrinking (vertex-disjoint) S-components in H' where S is a set of essential edges in H. (The operation of shrinking an S-component is defined in §4.1.)

LEMMA 8. *Let H' be an essential contraction of a 2-edge-connected graph H and let $H'_0, H'_1, \ldots, H'_k$ be a trace of Algorithm 1 for H'. Then, there is a trace $H_0, H_1, \ldots, H_k$ of Algorithm 1 for H such that H'_i is an essential contraction of H_i for $0 \leq i \leq k$. Hence $c(H') \leq c(H)$.*

Proof. Fix a 2-edge-connected graph H. Let H' be an essential contraction of H obtained from H by shrinking S-components in H. By Lemma 1 H' is 2-edge-connected and its complexity is indeed well defined. Fix a trace $H'_0, H'_1, \ldots, H'_k$ of Algorithm 1 for H'. We prove the lemma by induction on k. The base case $k = 0$ is clear: the graph H constitutes a trace of length 0 for H and H' is an essential contraction of H.

Assume that the claim holds if the trace for H' has length at most l. Let $H'_0, H'_1, \ldots, H'_{l+1}$ be a trace for H'. Let $H'_1 = T' + A'$ where T' is an optimal tree in H' and A' is a minimal augmentation for T' in H'. We may combine T' with spanning trees for the S-components in H to form a spanning tree T of H. By the optimality of T' and Corollary 2 the tree T contains a maximum number of essential edges in H that do not belong to S. Since T contains a spanning tree for each S-component, it also contains a maximum number of essential edges in S. Thus, T is an optimal tree in H. Let $A = A' \cup (S - E(T))$. By Corollary 2 and the fact that all edges in S are essential it follows that A is a minimal augmentation for T in H. The graph $T' + A'$ is an essential contraction of $T + A$. By the induction assumption there exists a trace $H_1, H_2, \ldots, H_{l+1}$ of Algorithm 1 for $T + A$ such that H'_i is an essential contraction of H_i for $1 \leq i \leq l + 1$. The claim of the lemma follows. □

THEOREM 13. *The graphs $F_0, F_1, \ldots$ form a sample for 2-edge-connectivity.*

Proof. The graph F'_i obtained from F_{i-1} by doubling the essential edges is certainly 2-edge-connected if F_{i-1} is 2-edge-connected. Moreover F'_i is an essential contraction of F_i. Thus F_i is 2-edge-connected if F_{i-1} is 2-edge-connected. Since F_0 is 2-edge-connected all F_i's are 2-edge-connected. Hence the complexity of F_i is well defined. We prove by induction on i the following two statements: (1) $c(F_i) \geq i$; (2) F_i has a spanning tree containing all redundant edges of F_i.

Statement (1) trivially holds for $i = 0$. Statement (2) holds for $i = 0$ since all edges in F_0 are essential. Assume inductively that the claim holds for $i = j - 1$. Recall that each vertex in F_{j-1} corresponds to a cycle in F_j; we refer to such a cycle in F_j as an F_j-cycle. All edges in F_j-cycles are essential in F_j since they have an endpoint of degree 2. All other edges in F_j are redundant in F'_j. Since F'_j is an essential contraction of F_j, Corollary 2 implies that these edges are redundant in F_j as well. By the construction of F_j no two redundant edges are incident on the same vertex. Hence there is a spanning tree in F_j containing all redundant edges of F_j, establishing statement (2).

To prove (1), fix a spanning tree T_{j-1} in F_{j-1} containing all redundant edges in F_{j-1}. Thus, $F_{j-1} = T_{j-1} + A_{j-1}$ where A_{j-1} is a minimal augmentation for T_{j-1} in F_{j-1}. We can combine T_{j-1} with spanning trees for the F_j-cycles (obtained by deleting an edge in each F_j-cycle) to form a spanning tree T_j of F_j. The tree T_j is an optimal tree in F_j since it contains a maximum number of edges in F_j-cycles. Let A_j be the edges in A_{j-1} as well as the edges in the F_j-cycles that are not in T_j. With Corollary 2 we see that A_j is a minimal augmentation for T_j in F_j. The graph F_{j-1} is an essential contraction of $T_j + A_j$. By the induction assumption and the previous lemma $c(T_j + A_j) \geq j - 1$. We conclude that $c(F_j) \geq j$. □

LEMMA 9. *Let n_i, m_i, and e_i denote the number of vertices, edges, and essential edges, respectively, in $F_i (i \geq 0)$. These quantities satisfy the following recurrence relations:*

$$n_{i+1} = 4(m_i + n_i),$$
$$m_{i+1} = m_i + e_i + n_{i+1},$$
$$e_{i+1} = n_{i+1},$$

and initial conditions $n_0 = m_0 = e_0 = 2$. Thus, $n_i = 4 \cdot 9^{i-1}$ and $m_i = 5 \cdot 9^{i-1}$ for $i > 0$.

COROLLARY 8. *For 2-edge-connectivity, there exists a function* $f(n) = \Omega(\log n)$ *such that there is a graph on* n *vertices of complexity* $f(n)$ *for any* $n \geq 1$.

Proof. To construct a graph of complexity $\Omega(\log n)$ with exactly n vertices, start with F_i where i is the maximum integer such that $n_i \leq n$ and increase the number of vertices in F_i to n by repeatedly subdividing an essential edge. The resulting graph has the same complexity as F_i, namely $\Omega(\log n)$. □

Let us now turn our attention to biconnectivity. Unfortunately, biconnectivity does not satisfy a condition similar to Lemma 8. We do, however, have the following result.

LEMMA 10. *If* G *is a graph with at least three vertices in which each vertex has degree* ≤ 3, *then* G *is biconnected if and only if* G *is 2-edge-connected.*

Proof. The only-if part is clear. Assume that G is 2-edge-connected and let u be a cutpoint in G. Hence, at least 2 blocks share u. Both of these blocks are 2-edge-connected and hence the degree of u is at least 4, contradicting the assumption that each vertex in G has degree ≤ 3. □

Let us call a graph in which each vertex has degree ≤ 3 a 3-*graph*. Let P = "2-edge-connectivity" and P' = "biconnectivity."

COROLLARY 9. *If a graph* H *is a 3-graph, then* $c_P(H) = c_{P'}(H)$.

Proof. By induction on $c_P(H)$. The induction base is clear with Lemma 10. Assume that the claim holds for $c_P(H) \leq k - 1$. Fix an H with $c_P(H) = k$. Thus, $H = T + A$ with $c_P(T + A) = k - 1$ where T is an optimal tree in H with respect to P and A is a minimal augmentation for T in H (with respect to P). By Lemma 10, an edge in H is P-redundant if and only if it is P'-redundant. Hence, T is an optimal tree in H with respect to P' and A is a minimal augmentation for T in H with respect to P'. Since $T + A$ is a 3-graph the induction hypothesis gives us $c_{P'}(T + A) \geq k - 1$ and hence $c_{P'}(H) \geq k = c_P(H)$. Similarly, one proves $c_P(H) \geq c_{P'}(H)$. We conclude that $c_P(H) = c_{P'}(H)$. □

Each F_i is a 3-graph. Since $n(F_i) \geq 3$ for $i \geq 1$, we have $c_{P'}(F_i) \geq i$ for $i \geq 1$ and the sequence $F_1, F_2, \ldots$ is a sample for biconnectivity. Thus, we get the following result.

COROLLARY 10. *For biconnectivity, there exists a function* $g(n) = \Omega(\log n)$ *such that there is a graph on* n *vertices with complexity* $g(n)$ *for any* $n \geq 3$.

Proof. The proof is similar to the proof of Corollary 8. □

Corollaries 8 and 10 establish that Algorithm 1 takes $\Omega(\log n)$ iterations for 2-edge-connectivity and biconnectivity in the worst case. Since each iteration of these algorithms takes $\Omega(n)$ time they require $\Omega(m + n \log n)$ time in the worst case. By Corollary 7 the same bound applies to Algorithm 3. Because of Theorem 7 this bound is tight for Algorithm 3. In [13] it is shown that redundant edges can be determined for 2-edge-connectivity and biconnectivity in linear time by modifying the linear-time triconnectivity algorithm of [21]. Thus the bound is tight for Algorithm 1 as well.

THEOREM 14. *Algorithms* 1 *and* 3 *require* $\Theta(m + n \log n)$ *operations in the worst case, both for 2-edge-connectivity and biconnectivity.*

6. Computing minimal spanning subgraphs in directed graphs. In this section we generalize the algorithms for undirected graphs to directed graphs. We provide general algorithms that compute a minimal spanning P-subgraph in a directed graph for any property P that is monotone and implies strong connectivity. For the special case of strong connectivity we obtain an algorithm that computes a minimal strongly connected spanning subgraph in time $O(m + n \log n)$. This algorithm is simpler (both in the sequential and the parallel implementation) than an earlier algorithm of [8] for this problem because it avoids the explicit computations of redundant edges. We prove that our algorithms require $\Theta(m + n \log n)$ time in the worst case for strong connectivity. By adapting the analysis we also get a $\Theta(m + n \log n)$ tight bound on the worst-case running time of an algorithm of [8] for finding a minimal strongly connected spanning subgraph. This answers an open question of [8].

6.1. High-level algorithm. Let G be a directed graph. A *forward branching* [8] rooted at a vertex x is a spanning subgraph of G whose underlying graph is a tree and in which x has in-degree zero and all other vertices have in-degree one. Throughout this section we shall assume that all forward branchings are rooted at a fixed vertex x of G.

It turns out that the development carried out in §3 for undirected graph properties carries over to digraphs with some modifications. The definitions of digraph property, P-graph, P-subgraph, and P-redundant and P-essential edges carry over from the undirected case without change. As for undirected graph properties we shall always assume that the property P is decidable. Conditions D1 and D2 correspond to conditions C1 and C2 in the undirected case (see §3). Note that condition D1 (monotonicity of P) is identical to C1.

D1. P is monotone, i.e., adding edges preserves property P.

D2. Any P-graph is strongly connected.

Algorithm 6 computes a minimal spanning P-subgraph of digraph G, i.e., a spanning P-subgraph of G in which all edges are P-essential. It has a structure similar to that of Algorithm 1; instead of computing a spanning tree in step (2.1), it computes a forward branching T_H rooted at a fixed vertex x of H (and containing a maximum number of P-essential edges). We call T_H an *optimal branching* in H (rooted at x) and A (see step (2.2) of Algorithm 6), as before, a *minimal augmentation* for T_H in H.

ALGORITHM 6. Computing a minimal spanning P-subgraph of G.

Input Digraph G with property P.

Output Minimal spanning P-subgraph H of G.

(1) $H := G$.

(2) While H has P-redundant edges, do:

(2.1) compute a forward branching T_H in H, rooted at x, with a maximum number of P-essential edges;

(2.2) compute a minimal subset A of edges in H such that $T_H + A$ has property P;

(2.3) $H := T_H + A$.

The following result is the analogue of Theorem 1. The proof is similar.

THEOREM 15. *Algorithm* 6 *computes a minimal spanning P-subgraph of G for any digraph property P satisfying* D1 *and* D2.

The following lemma will be needed to establish a logarithmic upper bound on the number of iterations of Algorithm 6. Let H be a digraph that contains a forward branching rooted at x. We say that an edge e of H is *f-redundant* in H if $H - e$ contains a forward branching rooted at x; an edge of H that is not f-redundant is *f-essential* in H.

LEMMA 11. *Let B be a set of f-redundant edges in H. There exists a forward branching rooted at x that contains at most half of the edges in B.*

Proof. Double the f-essential edges in H to obtain a graph H'. Let $X \subseteq V(H)$ with $x \in X$ and $X \neq V(H)$. Then there are at least two edges in H' from X to $V(H) - X$. By Edmonds' branching theorem [6] H' contains two edge-disjoint forward branchings rooted at x. Since no edge in B has been doubled, at least one of these two forward branchings contains at most half of the edges in B, yielding a forward branching in H rooted at x with the same property. □

THEOREM 16. *If P satisfies* D1 *and* D2, *then Algorithm* 6 *terminates after $O(\log n)$ iterations of the while-loop.*

Proof. Consider the start of an iteration of the while-loop. Let B denote the set of redundant edges in H. Each edge $e \in B$ is f-redundant in H since $H - e$ is strongly connected. By Lemma 11 there exists a forward branching rooted at x containing at most half of the edges in B. Thus an optimal branching in H contains at most half of the redundant

edges in H. Therefore at least half of the redundant edges in H are discarded or become essential in this iteration of the while-loop. The claim follows. □

As for undirected graph properties one can avoid the explicit computation of the P-essential (or P-redundant) edges by gradually building up a set of essential edges. The following algorithm uses this idea. We use m to denote the number of edges in G.

ALGORITHM 7. Computing a minimal spanning P-subgraph of G.
Input Digraph G with property P.
Output Minimal spanning P-subgraph H of G.
(1) $H := G$; $S := \emptyset$;
(2) for $i = 1$ to $\lceil \log m \rceil + 1$ do:
  (2.1) compute forward branching T_H in H rooted at x with maximum number of edges of S;
  (2.2) compute a minimal $A \subseteq E(H)$ such that $T_H + A$ has property P;
  (2.3) $H := T_H + A$; $S := S \cup A$.

THEOREM 17. *Algorithm 7 computes a minimal spanning P-subgraph of a digraph G with property P provided P satisfies* D1 *and* D2.

Proof. A straightforward induction over the iteration number shows that at the end of each iteration of the while-loop H is a spanning P-subgraph of G and the edges of S are essential in H.

We now show that H is minimal upon termination. Fix an iteration i of the while-loop. Let H_i (S_i) denote the graph H (the set S) at the beginning of iteration i and let R_i denote the set of edges in $E(H_i) - S_i$ that are f-redundant in H_i. To prove that the final graph H is minimal, it suffices to show that $|R_{i+1}| \leq \frac{1}{2}|R_i|$. Indeed, after $j = \lceil \log m \rceil + 1$ iterations we then have $R_j = \emptyset$, implying that all edges in $E(H_j) - S_j$ are f-essential in H_j and thus essential in H_j. Since all edges in S_j are essential as well, the graph H_j is minimal.

Any forward branching T in H_i must contain all edges of $E(H_i) - S_i$ that are f-essential in H. Thus a spanning tree T contains a maximum number of edges of S if and only if it contains a minimal number of edges in R_i. By Lemma 11 (with $B = R_i$) there exists a forward branching in H_i that contains at most half of the edges in R_i. We conclude that at least half of the edges in R_i are discarded or added to S in iteration i. Hence $|R_{i+1}| \leq \frac{1}{2}|R_i|$. □

6.2. Computing a minimal strongly connected spanning subgraph. In this subsection we adapt Algorithm 7 to strong connectivity. Before we do this we review the algorithm of [8] for computing a minimal strongly connected spanning subgraph (called *transitive compaction* in [8]).

The following definitions from [8] will be needed. Let G be a directed graph and $x \in V(G)$. An *inverse branching* rooted at x is a spanning subgraph of G whose underlying graph is a tree and in which x has out-degree zero and all other vertices have out-degree one. A *branching* is either a forward or an inverse branching. We assume that all branchings are rooted at a fixed vertex x of G. Let H be a subgraph of G. An *H-philic* (*H-phobic*) branching in G is one that has the greatest (smallest) number of edges in common with H over all branchings (rooted at x) in G.

In [8] the following algorithm is given for finding a minimal strongly connected spanning subgraph in a strongly connected digraph.

ALGORITHM 8. Computing a minimal strongly connected spanning subgraph H of G.
Input Strongly connected digraph G.
Output Minimal strongly connected spanning subgraph H of G.
(1) $H := G$;

(2) while H has redundant edges, do:
  (2.1) $R :=$ set of redundant edges in H;
  (2.2) $F :=$ R-phobic forward branching in H;
  (2.3) $I :=$ F-philic inverse branching in H;
  (2.4) $H := F \cup I$.

One iteration of the while-loop of Algorithm 8 can be performed in linear time (see [8] for details). Algorithm 8 is a special case of Algorithm 6 in the following sense: first, F (computed in step 2.2) is an optimal branching in H; second, $F \cup I = F \cup (I - F)$ and $I - F$ is a minimal augmentation for F in H. Thus, Theorem 16 (or a similar result of [8]) implies that Algorithm 8 runs in $O(m + n \log n)$ time.

We now adapt Algorithm 7 for strong connectivity so that it runs in $O(m + n \log n)$ time as well. A spanning tree T_H with a maximum number of edges of S is an S-philic forward branching. It can be computed in linear time using Edmonds' minimum weight branching algorithm as explained in [8]. To compute a minimal augmentation A for T_H, we compute a T_H-philic inverse branching I and set $A = E(I) - E(T_H)$. This can again be done in linear time using Edmonds' minimum weight branching algorithm [8]. Thus, Algorithm 7 computes a minimal strongly connected spanning subgraph in time $\Theta(m + n \log n)$.

This implementation of Algorithm 7 is simpler than Algorithm 8 since it does not require the redundant edges to be computed at each step (this is a fairly involved procedure). These simplifications are even more apparent in the parallel implementation of Algorithm 8. Indeed, most of [8] is concerned with developing a fairly involved parallel algorithm for computing the redundant edges. Algorithm 7 can be parallelized just as Algorithm 8. It does not require redundant edges to be computed (resulting in a much simpler implementation) while achieving the same parallel complexity.

As pointed out in [8], it is conceivable that Algorithm 8 terminates in a constant number of iterations of the while-loop, resulting in a linear worst-case running time. It would then be asymptotically faster than Algorithm 7, which always runs in time $\Theta(m + n \log n)$. In the remainder of this section we shall rule out this possibility by showing that Algorithm 8 requires $\Theta(m + n \log n)$ time in the worst case, thus answering a question of [8]. Because Algorithm 8 is a special case of Algorithm 6 this will imply a similar result for the worst-case time complexity of Algorithm 6 (for strong connectivity).

To analyze the worst-case time complexity of Algorithm 8, we first observe that this algorithm chooses a *minimum* augmentation for F at each step, i.e., a minimal augmentation of smallest size.

LEMMA 12. *Fix an iteration of the while-loop of Algorithm* 8. *The edges of I that are not in F form a minimum augmentation for F in H. Conversely, if A is a minimum augmentation for an R-phobic forward branching F in H, then $F + A$ is of the form $F \cup I$ where I is an F-philic inverse branching in H.*

Proof. The lemma follows routinely from the following two facts: $F + A$ is strongly connected if and only if it contains an inverse branching (rooted at x) and all branchings in H have the same number of edges. □

The previous lemma motivates the following definition. For a strongly connected digraph H a *minimum augmentation trace for H with respect to x* is a sequence $H_0, H_1, \ldots, H_k$ such that $H_0 = H$, $H_i \neq H_{i+1}$ for $0 \leq i < k$, and $H_i (0 < i \leq k)$ is of the form $T + A$ where T is an optimal branching in H_{i-1} (i.e., a forward branching with a minimum number of redundant edges) rooted at x and A is a *minimum* augmentation for T in H_{i-1}. The integer k is the *length* of the trace. Let $\hat{c}(H, x)$ denote the maximum length of a minimum augmentation trace for H with respect to x and let $\hat{c}(H)$ stand for $\max\{\hat{c}(H, x) : x \in V(H)\}$. Note that $\hat{c}(H)$ is the worst-case number of iterations of Algorithm 8 if we let an adversary choose a root and choose

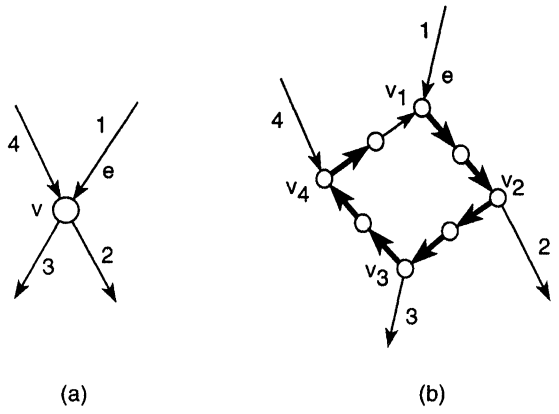


FIG. 4. (a) *Vertex $v \neq x$ in G of degree* 4; *the edge e belongs to branching $\hat{T}$. The labels on the edges are those assigned in step* 5. (b) *The cycle through the representatives of v together with the edges in F'_{i+1} corresponding to the edges labeled* 1–4 *in* (a). *The thick edges are those in the path $P(v)$ (rooted at v_1).*

in each iteration an R-phobic forward branching F and an F-philic inverse branching I. A minimum augmentation sample (for strong connectivity) is a sequence of digraphs $F'_0, F'_1, \ldots$ such that $\hat{c}(F'_i) \geq i$.

The procedure for constructing a minimum augmentation sample for strong connectivity is similar to that for constructing a sample for 2-edge-connectivity although it is more complicated. Let F'_0 be a directed cycle of length 2. We use x to denote an arbitrary fixed vertex in F'_i. We construct F'_{i+1} from F'_i as follows:

ALGORITHM 9. Computing a minimum augmentation sample for strong connectivity.

Input Digraph F'_i.

Output Digraph F'_{i+1}.

(1) Double the essential edges in F'_i. Call the resulting graph G.

(2) Let $\hat{T}$ be a forward branching in G rooted at x.

(3) For each vertex v in G of degree d, we add d new vertices $v_1, \ldots, v_d$ - called the *representatives of* v - to F'_{i+1} and connect them into a (directed) cycle; subdivide each edge on this cycle with a new vertex.

(4) For each vertex v in G of degree d, number the representatives of v as follows: fix a path $P(v)$ that spans the cycle constructed for v in step 3 and that is rooted at a representative v_i of v. Assign labels $1, 2, \ldots, d$ to the representatives of v in increasing order of their distance from v_i on $P(v)$. (Hence, v_i is labeled 1.)

(5) For a vertex v in G of in-degree p and out-degree q, label the $p + q$ edges incident on v as follows: if $v \neq x$, assign label 1 to the unique incoming edge that belongs to $\hat{T}$ (constructed in step 2); the outgoing edges receive labels $2 \ldots 1+q$ and the labels $2+q \ldots p+q$ are assigned to the other incoming edges. If $v = x$, then assign the labels $1 \ldots q$ to the outgoing edges and the labels $1 + q \ldots p + q$ to the incoming edges.

(6) For each edge (u, v) in G that has label i at u and label j at v (with respect to the labeling of step 5), add an edge from the representative of u with label i to the representative of v with label j in F'_{i+1} (with respect to the labeling defined in step 4).

Figure 4 illustrates steps 3-6 of Algorithm 9. Figure 5 shows the first three graphs in a possible sample. Note that these graphs are orientations of the corresponding graphs in the sample for 2-edge-connectivity (see Fig. 3).

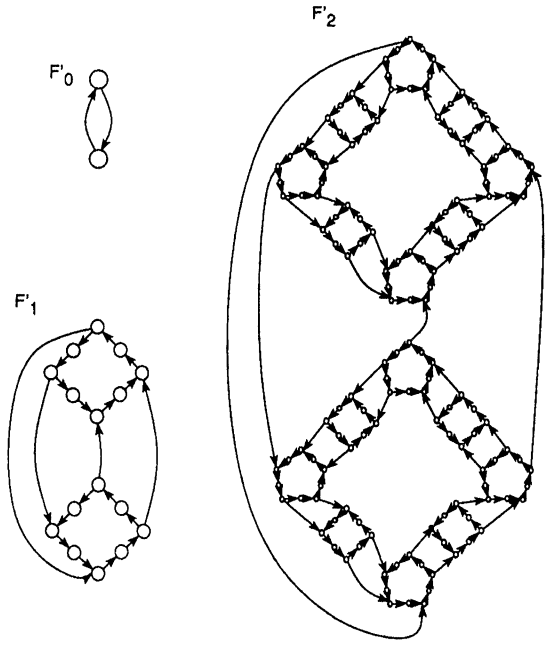


FIG. 5. F'_0, F'_1, *and* F'_2.

The next two results will be needed to prove that the graphs F'_i form a minimal augmentation sample for strong connectivity.

LEMMA 13. *Let H be a directed graph and $S \subseteq E(H)$. Let H' be obtained from H by collapsing in H the vertex sets of the strong components of $(V(H), S)$. Then H is strongly connected if and only if H' is strongly connected. Furthermore, if both graphs are strongly connected, then an edge $e \in E(H')$ is essential in H' if and only if it is essential in H.*

Proof. A straightforward adaptation of the proofs of the corresponding claims for shrinking S-components in a 2-edge-connected graph (see Lemma 1 and Corollary 2). □

We say that H' is an *essential contraction* of a strongly connected graph H (with respect to a subset S of $E(H)$) if H' is obtained from H by collapsing in H the vertex sets of the strong components of $(V(H), S)$ and the edges in S are essential in H. If $v \in V(H)$ belongs to a strong component that is collapsed into a new vertex z, then we say that v is collapsed into z. If $v \in V(H) \cap V(H')$ then we say that v is collapsed into v.

LEMMA 14. *Let H' be an essential contraction of a strongly connected graph H and let $H'_0, H'_1, \ldots, H'_k$ be a minimum augmentation trace for H' with respect to a vertex x. Then there is a minimum augmentation trace $H_0, H_1, \ldots, H_k$ for H with respect to any vertex y that is collapsed into x such that H'_i is an essential contraction of H_i for $0 \leq i \leq k$. Hence $\hat{c}(H') \leq \hat{c}(H)$.*

Proof. Fix a strongly connected graph H. Let H' be an essential contraction of H with respect to a set S of essential edges in H. By Lemma 13 H' is strongly connected. We prove the lemma by induction on k, the length of the trace for H'. If $k = 0$, then the minimum augmentation trace for H' consists of H' itself. The graph H by itself constitutes a minimum augmentation trace of length 0 for H (with respect to an arbitrary vertex). The base case follows.

Let $H'_0, H'_1, \ldots, H'_k$ be a minimum augmentation trace for H' with respect to a vertex x with $k > 0$. Let $H'_1 = T' + A'$ where T' is an optimal branching in $H'_0 = H'$ rooted

at x and A' is a minimum augmentation for T' in H'. We may combine T' with forward branchings for the strong components of $(V(H), S)$ to form a forward branching T of H rooted at an arbitrary vertex y of H collapsed into x. By the optimality of T' and Lemma 13 the forward branching T contains a maximum number of essential edges in H that do not belong to S (among all forward branchings in H rooted at y). Since T contains a forward branching for each strong component of $(V(H), S)$, T also contains a maximum number of edges of S (which are essential in H). Thus, T is an optimal branching in H rooted at y. Let $A = A' \cup (S - E(T))$. By Lemma 13 and the fact that all edges in S are essential it follows that A is a minimum augmentation for T in H. The graph $T' + A'$ is an essential contraction of $T + A$. By the induction assumption there is a minimum augmentation trace $H_1, H_2, \ldots, H_k$ for $T + A$ with respect to any vertex z that is collapsed into x such that H_i' is an essential contraction of H_i for $1 \leq i \leq k$. We prefix this minimum augmentation trace with H to get a minimum augmentation trace $H, H_1, \ldots, H_k$ for H that has the claimed property. □

THEOREM 18. *We have* $\hat{c}(F_i') \geq i$ *for all* $i \geq 0$.

Proof. We prove the following stronger statement: for each $i \geq 0$ there is a vertex x in F_i' such that $\hat{c}(F_i', x) \geq i$ and the edge set of F_i' can be partitioned into a forward branching rooted at x and a set of essential edges. We show this by induction on i.

The base case clearly holds. Assume the statement holds for F_i'. Let G be the digraph constructed by doubling the essential edges in F_i' (step 1 of Algorithm 9). By the induction assumption the graph F_i' is of the form $T + A$ where T is a forward branching in F_i' rooted at a vertex x and A is a set of essential edges in F_i'. Since every edge of G is redundant, T is an optimal branching in G rooted at x. Furthermore, A is a minimum augmentation for T in G. To see this, fix an edge e in A. Let e' be the edge parallel to e in G. By the definition of G and the fact that e is essential in F_i', it follows that any minimal augmentation for T in G has a nonempty intersection with $\{e, e'\}$. Thus, the set A constitutes a minimum augmentation for T in G. Hence, $\hat{c}(G, x) \geq i + 1$. Note that G is an essential contraction of F_{i+1}'. By Lemma 14 we have $\hat{c}(F_{i+1}, y) \geq i + 1$ for any vertex y in the subgraph of F_{i+1}' collapsed into x.

We now show that the edge set of F_{i+1}' can be partitioned into a forward branching rooted at one such vertex y and a set of essential edges. Let y be the root of the path $P(x)$ used in step 4 of Algorithm 9, i.e., y is that representative of x that is labeled 1 in step 4 of Algorithm 9. Let B denote the set of edges in F_{i+1}' that also belong to G. By Lemma 13 these edges are exactly the redundant edges in F_{i+1}'. It suffices to show that there exists a forward branching in F_{i+1}' rooted at y and containing all the edges of B.

For any vertex $w \neq y$ in F_{i+1}' that has no incoming edge in B, define $e(w)$ to be the unique edge in $P(v)$ whose head is w. Let B' be the set $\{e(w) : w \in V(F_{i+1}'), w \neq y$ and w has no incoming edge in $B\}$. We shall now prove that every vertex in F_{i+1}' is reachable from y by edges in $B \cup B'$. Since each vertex in F_{i+1}' other than y has exactly one incoming edge in $B \cup B'$ and y has none, this implies that the edges in $B \cup B'$ form a forward branching in F_{i+1}' rooted at y; hence, F_{i+1}' can be partitioned into a forward branching rooted at y and a set of essential edges.

Let us call a path in F_{i+1}' all of whose edges are in $B \cup B'$ a *good path*. We denote the root of $P(v)$ by $r(v)$ for any v in G (see step 4 of Algorithm 9). We first show that there is a good path from y to the root $r(v)$ of $P(v)$ in F_{i+1}' for any v in G. Let *depth*(v) denote the depth of v in $\hat{T}$ (constructed in step 2 of Algorithm 9), i.e., the number of edges on the unique path from x to v in $\hat{T}$. We prove the claim by induction on *depth*(v). If *depth*$(v) = 0$, then $v = x$ and $r(v) = y$; hence, the base case holds. Assume inductively that the claim holds if *depth*$(v) = k$. Now let *depth*$(v) = k + 1$. Let u be the father of v in $\hat{T}$. By the induction assumption there is a good path from y to the root $r(u)$ of $P(u)$. Let $(w, r(v))$ be the edge in F_{i+1}' corresponding to edge (u, v) of $\hat{T}$. From steps 4, 5, and 6 of Algorithm 9 it follows that

there is a good path in $P(u)$ from $r(u)$ to w. We can assemble the good paths from y to $r(u)$ and from $r(u)$ to w together with the edge $(w, r(v))$ into a good path from y to $r(v)$ in F'_{i+1}.

Now consider the case where a vertex w in F'_{i+1} lies on the path $P(v)$ of some vertex v in G but is different from the root of $P(v)$. Vertex w is reachable on $P(v)$, using only edges of B', either from the root of $P(v)$ or from a vertex w' on $P(v)$ that has an incoming edge in B. In the former case we are done. In the latter case let w'' be the tail of the edge of B in F'_{i+1} whose head is w'. With the labeling in step 5 of Algorithm 9 it follows that w'' is reachable from the root of its path using only edges of B'. We conclude that there is a good path from y to w. □

THEOREM 19. *Algorithm* 8 *requires* $\Omega(\log n)$ *iterations in the worst case. Thus, its worst-case time complexity is* $\Theta(m + n \log n)$.

Proof. The graphs F'_i are orientations of the graphs F_i in the sample for 2-edge-connectivity and thus have the same size. With Lemma 9 and Theorem 18 we have $\hat{c}(F'_i) = \Omega(\log n(F'_i))$. One can construct for each $n > 1$ a digraph G on n vertices with $\hat{c}(G) = \Omega(\log n)$ by subdividing edges in the graph F'_i where i is the largest integer with $n(F'_i) \leq n$. Since each iteration of Algorithm 8 requires time $\Omega(n)$, we infer that Algorithm 8 requires $\Omega(m + n \log n)$ time in the worst case. This bound is tight because it is shown in [8] that the algorithm terminates in time $O(m + n \log n)$. □

We now turn our attention to Algorithm 6. For a strongly connected digraph H a *trace for* H *with respect to* x is a sequence $H_0, H_1, \ldots, H_k$ such that $H_0 = H, H_i \neq H_{i+1}$ for $0 \leq i < k$, and $H_i (0 < i \leq k)$ is of the form $T + A$ where T is an optimal branching in H_{i-1} rooted at x and A is a minimal augmentation for T in H_{i-1}. Let $c(H)$ denote the maximum length of a trace for H (maximum taken over all vertices). Since each minimum augmentation trace for H is also a trace for H with respect to the same vertex, Theorem 18 implies that $c(F'_i) \geq i$. The following theorem is an easy corollary of this fact.

THEOREM 20. *Algorithm* 6 *requires* $\Omega(\log n)$ *iterations in the worst case. Thus, its worst-case time complexity is* $\Theta(m + n \log n)$.

7. Concluding remarks. In this paper we have given linear time algorithms for computing a minimal biconnected spanning subgraph and a minimal 2-edge-connected spanning subgraph. We have also provided a general framework for computing minimal spanning subgraphs with respect to various graph properties. These results should be useful in deriving algorithms for computing minimal spanning subgraphs for other graph properties, e.g., k-vertex- or k-edge-connectivity for $k > 2$.

In the context of directed graphs we leave open the question whether there is a linear time algorithm for computing a minimal strongly connected spanning subgraph. We have shown in this paper that several natural algorithms for this problem achieve a worst-case running time of $\Theta(m + n \log n)$. The results in this paper suggest that it may be possible to achieve linear time by combining the basic algorithms with various contraction operations (this approach has worked for 2-edge-connectivity and biconnectivity). Unfortunately we proved in [14] (using a fairly involved construction) that this approach will not improve the worst-case running time if we collapse cycles and contract chains only. A new approach seems necessary.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] J. A. BONDY AND U. S. R. MURTY, *Graph Theory with Applications*, North-Holland, Amsterdam, 1976.

[3] F. CHUNG AND R. L. GRAHAM, private communication, 1977; cited in [7].

[4] R. COLE AND U. VISHKIN, *Approximate parallel scheduling, part* I: *The basic technique with applications to optimal parallel list ranking in logarithmic time*, SIAM J. Comput., 17 (1988), pp. 128–142.

[5] E. DAHLHAUS, M. KARPINSKI, AND P. KELSEN, *An efficient parallel algorithm for finding a maximal independent set in a hypergraph of dimension* 3, Inform. Process. Letters, 42 (1992), pp. 309-313.

[6] J. EDMONDS, *Edge-disjoint branchings*, in Combinatorial Algorithms, Algorithmic Press, New York, 1973, pp. 91-96.

[7] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, CA, 1979.

[8] P. GIBBONS, R. M. KARP, V. RAMACHANDRAN, D. SOROKER, AND R. TARJAN, *Transitive compaction in parallel via branchings*, J. Algorithms, 12 (1991), pp. 110-125.

[9] M. GOLDBERG AND T. SPENCER, *A new parallel algorithm for the maximal independent set problem*, SIAM J. Comput., 18 (1989), pp. 419-427.

[10] X. HAN, *An Algorithmic Approach to Extremal Graph Problems*, Ph.D. Thesis, Department of Computer Science, Princeton University, Princeton, NJ, June 1991.

[11] R. M. KARP AND V. RAMACHANDRAN, *Parallel algorithms for shared-memory machines*, in Handbook of Theoretical Computer Science, Vol. A, J. van Leeuwen, ed., MIT Press, Cambridge, MA, Elsevier, 1990, pp. 869-941.

[12] R. M. KARP, E. UPFAL, AND A. WIGDERSON, *The complexity of parallel search*, J. Comput. System Sci., 36 (1988), pp. 225-253.

[13] P. KELSEN AND V. RAMACHANDRAN, *On finding minimal two-connected subgraphs*, J. Algorithms, 18 (1995), pp. 1-49.

[14] ———, *The complexity of finding minimal spanning subgraphs*, Tech. Report TR-91-17, Department of Computer Sciences, University of Texas, Austin, TX, 1991.

[15] J. M. LEWIS AND M. YANNAKAKIS, *The node-deletion problem for hereditary properties is NP-complete*, J. Comput. System. Sci., 20 (1980), pp. 219-230.

[16] M. LUBY, *A simple parallel algorithm for the maximal independent set problem*, SIAM J. Comput., 15 (1986), pp. 1036-1053.

[17] G. MILLER AND J. H. REIF, *Parallel tree contraction and its applications*, in Proc. 26th Ann. Symp. on Foundations of Comp. Sci., IEEE Press, New York, 1985, pp. 478-489.

[18] H. NAGAMOCHI AND T. IBARAKI, *Linear time algorithms for finding a sparse k-connected spanning subgraph of a k-connected graph*, Algorithmica, 7 (1992), pp. 583-596.

[19] M. D. PLUMMER, *On minimal blocks*, Trans. Amer. Math. Soc., 134 (1968), pp. 85-94.

[20] V. RAMACHANDRAN, *Fast parallel algorithms for reducible flow graphs*, in Concurrent Computations: Algorithms, Architecture and Technology, S. K. Tewksbury, B. W. Dickinson, and S. C. Schwartz, eds., Plenum Press, New York, 1988, pp. 117-138; see also *Fast and processor-efficient parallel algorithms for reducible flow graphs*, Tech. Report ACT-103, Coordinated Science Laboratory, University of Illinois, Urbana, IL, November 1988.

[21] ———, *Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity*, in Synthesis of Parallel Algorithms, J. Reif, ed., Morgan Kaufmann, San Mateo, 1993, pp. 275-340.

[22] R. TARJAN, *Depth first search and linear graph algorithms*, SIAM J. Comput., 1 (1972), pp. 146-160.

[23] W. TUTTE, *Graph Theory*, Addison-Wesley, Reading, MA, 1984.

[24] M. YANNAKAKIS, *Node- and edge-deletion NP-complete problems*, in Proc. 10th Ann. ACM Symp. on Theory of Computing, New York, 1978, pp. 253-264.